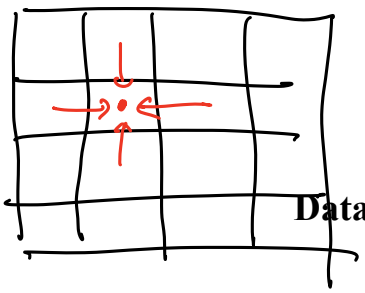


Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures

Tom Henretty[1], Kevin Stock[1], Louis-Noël Pouchet[1], Franz Franchetti[2], J. Ramanujam[3], and P. Sadayappan[1]

[1] The Ohio State University [2] Carnegie Mellon University [3] Louisiana State University

Abstract. Stencil computations are at the core of applications in many domains such as computational electromagnetics, image processing, and partial differential equation solvers used in a variety of scientific and engineering applications. Short-vector SIMD instruction sets such as SSE and VMX provide a promising and widely available avenue for enhancing performance on modern processors. However a fundamental memory stream alignment issue limits achieved performance with stencil computations on modern short SIMD architectures. In this paper, we propose a novel data layout transformation that avoids the stream alignment conflict, along with a static analysis technique for determining where this transformation is applicable. Significant performance increases are demonstrated for a variety of stencil codes on several modern processors with SIMD capabilities.

1 Introduction

Short vector SIMD extensions are included in all major high-performance CPUs. While ubiquitous, the ISA and performance characteristics vary from vendor to vendor and across hardware generations. For instance, Intel has introduced SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, and LRBni ISA extensions over the years. With every processor, the latency and throughput numbers of instructions in these extensions change. IBM, Freescale, and Motorola have introduced AltiVec, VMX, VMX128, VSX, Cell SPU, PowerXCell 8i SPU SIMD implementations. In some instances (RoadRunner, BlueGene/L), custom ISA extensions were designed since the supercomputing installation was big enough to warrant such an investment. These extensions provide from 2-way adds and multiplies up to 16-way fused multiply-add operations, promising significant speed-up. It is therefore important to optimize for these extensions.

Stencil computations represent an important class, occuring in many image processing applications, computational electromagnetics and solution of PDEs using finite difference or finite volume discretizations. There has been considerable recent interest in optimization of stencil computations [7], [6], [16], [17], [27], [26], [11], [37], [10], [4], [9], [40], [38], [41], [8], [39], [43], [25], [21], [34]. In this paper, we focus on the problem of optimizing single-core performance on modern processors with SIMD instruction sets. Stencil computations are readily expressible in a form with vectorizable innermost loops where arrays are accessed at unit stride. However, as elaborated upon later, there is a fundamental performance limiting factor with all current short-vector SIMD instruction sets such as SSE, AVX, VMX, etc. We formalize the problem through the abstraction of stream alignment conflicts. We note that the alignment conflict issue we formulate and solve in this paper pertains to algorithmic alignment constraints and is distinctly different from the previously studied topic of efficient code generation on SIMD architectures with hardware alignment constraints [22, 12, 44, 13]. We address the problem

of resolving stream alignment conflict through the novel use of a nonlinear data layout transformation and develop a compiler framework to identify and suitably transform the computations.

```
for (t = 0; t < T; ++t) {
  for (i = 0; i < N; ++i)
    for (j = 1; j < N+1; ++j)
S1:   C[i][j] = A[i][j] + A[i][j-1];
  for (i = 0; i < N; ++i)
    for (j = 1; j < N+1; ++j)
S2:   A[i][j] = C[i][j] + C[i][j-1];
}
```

Performance:		
	AMD Phenom	1.2 GFlop/s
	Core2	3.5 GFlop/s
	Core i7	4.1 GFlop/s

(a) Stencil code

```
for (t = 0; t < T; ++t) {
  for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
S3:   C[i][j] = A[i][j] + B[i][j];
  for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
S4:   A[i][j] = B[i][j] + C[i][j];
}
```

Performance:		
	AMD Phenom	**1.9** GFlop/s
	Core2	**6.0** GFlop/s
	Core i7	**6.7** GFlop/s

(b) Non-Stencil code

Fig. 1. Example to illustrate addressed problem: The stencil code (a) has much lower performance than the non-stencil code (b) despite acccessing 50% fewer data elements.

We use a simple example to illustrate the problem addressed. Consider the two sets of loop computations in Fig. 1(a) and Fig. 1(b), respectively. With the code shown in Fig. 1(a), for each of the two statements, $2 \times N^2 + N$ distinct data elements are referenced, with N^2 elements of C and $N^2 + N$ elements of A being referenced in S1, and N^2 elements of A and $N^2 + N$ elements of C being referenced in S2. With the code shown in Fig. 1(b), each of S3 and S4 access $3 \times N^2$ distinct data elements, so that the code accesses around 1.5 times as many data elements as the code in Fig. 1(a). Both codes compute exactly the same number ($2 \times N^2$) of floating point operations. Fig. 1 shows the achieved single-core performance of the two codes on three different architectures, compiled using the latest version of ICC with auto-vectorization enabled. It may be seen that on all systems, the code in Fig. 1(b) achieves significantly higher performance although it requires the access of roughly 50% more data elements.

As explained in the next section, the reason for the lower performance of the stencil code in Fig. 1(a) is that adjacent data elements (stored as adjacent words in memory) from arrays A and C must be added together, while the data elements that are added together in the code of Fig. 1(b) come from independent arrays. In the latter case, we can view the inner loop as representing the addition of corresponding elements from two independent streams `B[i][0:N-1]` and `C[i][0:N-1]`, but for the former, we are adding shifted versions of data streams: `A[i][0:N-1]`, `A[i][1:N]`, `C[i][0:N-1]`, and `C[i][1:N]`. Loading vector registers in this case requires use of either (a) redundant loads, where a data element is moved with a different load for each distinct vector register position it needs to be used in, or (b) load operations followed by additional inter- and intra-register movement operations to get each data element into the different vector register slots where it is used. Thus the issue we address is distinctly different from the problem of hardware alignment that has been addressed in a number of previous works. The problem we address manifests itself even on architectures where hardware alignment is not necessary and imposes no significant penalty (as for example the recent Intel Core i7 archirecture).

In this paper, we make the following contributions:

- We identify a fundamental performance bottleneck for stencil computations on all short-vector SIMD architectures and develop a novel approach to overcoming the problem via data layout transformation.

– We formalize the problem in terms of stream alignment conflicts and present a compile-time approach to identifying vectorizable computations that can benefit from the data layout transformation.
– We present experimental results on three target platforms that demonstrate the effectiveness of the transformation approach presented in this paper.

To the best of our knowledge, this is the first work to identify and formalize this problem with *algorithmic* stream alignment conflicts and provide a solution to it.

The paper is structured as follows. In the next section, we elaborate on the addressed problem. In Sec. 3, we provide an overview of the data layout transformation approach through examples. Sec. 4 presents the formalization and compile-time analysis for detecting stream alignment conflicts. Sec. 5 presents experimental methodology and results. Related work is covered in Sec. 6 and we conclude in Sec. 7.

2 Background and Overview of Approach

2.1 Illustrative Example

The reason for the significant performance difference between the two codes shown in Sec. 1, is that one of them (Fig. 1(a)) exhibits what may be called *stream alignment conflict*, while the other (Fig. 1(b)) is free of such conflicts. When stream alignment conflicts exist, the compiler must generate additional loads or inter-register data movement instructions (such as shuffles) in order to get interacting data elements into the same vector register positions before performing vector arithmetic operations. These additional data movement operations cause the performance degradation seen in Fig. 1. Fig. 2 shows the x86 assembly instructions generated by the Intel ICC compiler on a Core 2 Quad system for code similar to that in Fig. 1(a), along with an illustration of the grouping of elements into vectors. The first four iterations of inner loop j perform addition on the pairs of elements `(B[0],B[1])`, `(B[1],B[2])`, `(B[2],B[3])`, and `(B[3],B[4])`. Four single-precision floating-point additions can be performed by a single SSE SIMD vector instruction (`addps`), but the corresponding data elements to be added must be located at the same position in two vector registers. To accomplish this, ICC first loads vectors `B[0:3]` and `B[4:7]` into registers xmm1 and xmm2. Next, a copy of vector `B[4:7]` is placed into register xmm3. Registers xmm1 and xmm3 are then combined with the `palignr` instruction to produce the unaligned vector `B[1:4]` in register xmm3. This value is used by the `addps` instruction to produce an updated value for `A[0:3]` in xmm3. Finally, an aligned store updates `A[0:3]`. The vector `B[4:7]` in register xmm2 is ready to be used for the update of `A[4:7]`.

2.2 Stream Alignment Conflict

We now use a number of examples to explain the issue of stream alignment conflicts. Before we proceed, we reiterate that the issue we address is a more fundamental algorithmic data access constraint and is not directly related to the hardware alignment restrictions and penalties on some SIMD instruction set architectures. For example, on IBM Power series architectures using the VMX SIMD ISA, vector loads and stores must be aligned to quadword boundaries. On x86 architectures, unaligned vector loads/stores are permitted but may have a significant performance penalty, as on the Core 2 architecture. On the more recent Core i7 x86 architecture, there is very little penalty for unaligned loads versus aligned loads. The performance difference on the Core i7 shown in Fig. 1, however,

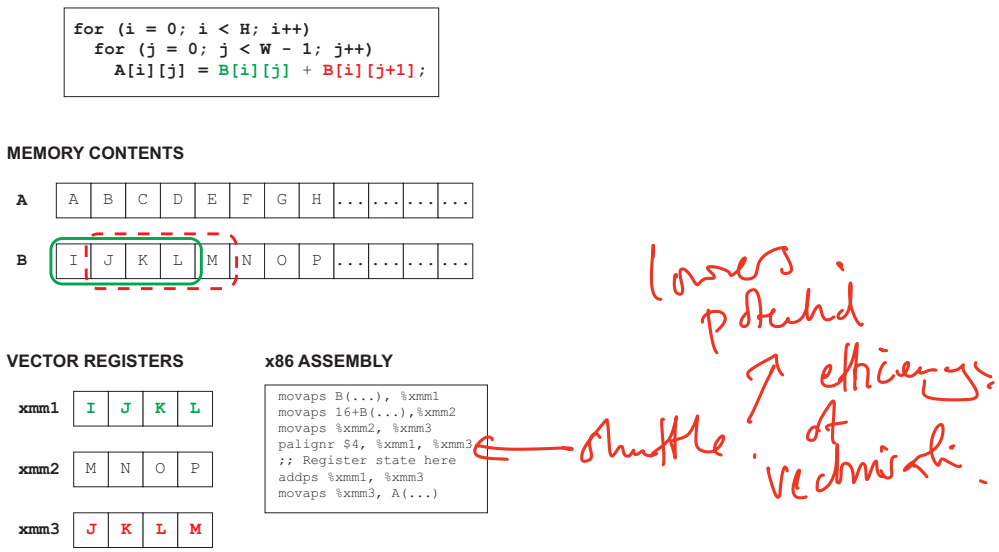


Fig. 2. Illustration of additional data movement for stencil operations

provides evidence that the problem we address is a more fundamental algorithmic alignment issue. While hardware alignment restrictions/penalties on a target platform may exacerbate the problem with stream alignment conflicts, the problem exists even if an architecture has absolutely no restrictions/penalties for unaligned loads/stores. The work we present in this paper thus addresses a distinctly different problem than that addressed by many other works on optimizing code generation for SIMD vector architectures with hardware alignment constraints.

Vectorizable computations in innermost loops may be viewed as operations on streams corresponding to contiguous data elements in memory. The computation in the inner loop with statement S1 in Fig. 1(b) performs the computation `C[i][0:N-1] = A[i][0:N-1]+ B[i][0:N-1]`, i.e. the stream of N contiguous data elements `A[i][0:N-1]` is added to the stream of N contiguous elements `B[i][0:N-1]` and the resulting stream is stored in `C[0][0:N-1]`. In contrast, the inner loop with statement S1 in Fig. 1(a) adds `A[i][1:N]` and `A[i][0:N-1]`, where these two streams of length N are subsets of the same stream `A[i][0:N]` of length N+1, but one is shifted with respect to the other by one element. When such shifted streams are computed upon using current short-vector SIMD architectures, although only N+1 distinct elements are accessed, essentially $2 \times N$ data moves are required (either through additional explicit loads or inter-register data movement operations like shuffles). The extra moves are required because of the inherent characteristic of short-vector SIMD instructions that only elements in corresponding slots of vector registers can be operated upon.

While in the example of Fig. 1(a) the stream alignment conflict is apparent because the misaligned streams arise from two uses of the same array in the same statement, the same underlying conflict may arise more indirectly, as illustrated by the example of Fig. 3.

In Fig. 3(a), the stream computations corresponding to the inner loop *j* are `A[i][0:N] = B[i][0:N]+C[i][0:N]` for S3 and `D[i][0:N] = A[i][0:N]+C[i][0:N]` for S4. Streams `A[i][0:N]` and `C[i][0:N]` are used twice, but all accesses are mutually aligned.

```
      for (i = 0; i < N+1; ++i)
        for (j = 0; j < N+1; ++j) {
S3:       A[i][j] = B[i][j] + C[i][j];
S4:       D[i][j] = A[i][j] + C[i][j];
      }
```

(a) No stream alignment conflict

```
      for (i = 0; i < N+1; ++i)
        for (j = 0; j < N+1; ++j) {
S5:       A[i][j] = B[i][j] + C[i][j];
S6:       D[i][j] = A[i][j] + C[i][j+1];
      }
```

(b) Stream alignment conflict exists

```
      for (i = 0; i < N+1; ++i)
        for (j = 0; j < N+1; ++j) {
S7:       A[i][j] = B[i][j] + C[i][j];
S8:       C[i][j] = A[i][j] + D[i][j+1];
      }
```

(c) No stream alignment conflict

Fig. 3. Illustration of indirect occurrence of stream alignment conflict

With the example shown in Fig. 3(b), however, the stream computations corresponding to the inner *j* loop are `A[i][0:N] = B[i][0:N]+C[i][0:N]` for S5 and `D[i][0:N] = A[i][0:N]+C[i][1:N+1]` for S6. Here we have a fundamental stream alignment conflict. If `A[i][0:N]` and `C[i][0:N]` are aligned together for S5, there is a misalignment between `A[i][0:N]` and `C[i][1:N+1]` in S6. Finally, there is no stream alignment conflict in Fig. 3(c) since the same alignment of `A[i][0:N]` with `C[i][0:N]` is needed for both S7 and S8.

The abstraction of stream alignment conflicts is applicable with more general array indexing expressions, as illustrated by the examples in Fig. 4. In Fig. 4(a), there is no stream alignment conflict. In S9, streams `B[i][i:i+N]` and `B[i+1][i:i+N]` (the second and third operands) need to be aligned. Exactly the same alignment is needed between these streams in S10 (the first operand `B[i][i+1:i+N+1]` and the second operand `B[i+1][i+1:i+N+1]`).

```
    for (i = 1; i < N+1; ++i)
      for (j = 0; j < N+1; ++j) {
S9:     A[i][j]   = B[i-1][i+j] +
                    B[ i ][i+j] +
                    B[i+1][i+j];
S10:  A[i+1][j] = B[ i ][i+j+1] +
                    B[i+1][i+j+1] +
                    B[i+2][i+j+1];
    }
```

(a) No stream alignment conflict

```
    for (i = 1; i < N+1; ++i)
      for (j = 0; j < N+1; ++j) {
S11:  A[i][j]   = B[i-1][i+j+1] +
                    B[ i ][ i+j ] +
                    B[i+1][i+j+1];
S12:  A[i+1][j] = B[ i ][i+j+2] +
                    B[i+1][i+j+1] +
                    B[i+2][i+j+2];
    }
```

(b) Stream alignment conflict exists

Fig. 4. Illustration of stream alignment conflict with general array indexing expressions

In contrast, in Fig. 4(b) a fundamental algorithmic stream alignment conflict exists with the reused streams in S11 and S12. In S11, the stream `B[i][i:i+N]` (second operand) must be aligned with `B[i+1][i+1:i+N+1]` (third operand), but in S12 stream `B[i][i+2:i+N+2]` (first operand) must be aligned with `B[i+1][i+1:i+N+1]` (second operand). Thus the required alignments between the reused streams in S11 and S12 are inconsistent — a +1 shift of `B[i][x:y]` relative to `B[i+1][x:y]` is needed for S11 but a -1 shift of `B[i][x:y]` relative to `B[i+1][x:y]` is needed for S12 i.e., a stream alignment conflict exists.

A formalization and algorithm for compile-time characterization of the occurrences of stream alignment conflicts is developed in Sec. 4. Before providing those details,

we present a novel approach through data layout transformation to avoid performance degradation due to stream alignment conflicts.

3 Data Layout Transformation

In this section, we show how the poor vectorization resulting from stream alignment conflicts can be overcome through data layout transformation. As explained in the previous section using Fig. 2, the main problem is that adjacent elements in memory map to adjacent slots in vector registers, so that vector operations upon such adjacent elements cannot possibly be performed without either performing another load operation from memory to vector register or performing some inter-register data movement. The key idea behind the proposed data layout transformation is for potentially interacting data elements to be relocated so as to map to the same vector register slot.

3.1 Dimension-Lifted Transposition

We explain the data layout transformation using the example in Fig. 5. Consider a one dimensional array Y with 24 single-precision floating point data elements, shown in Fig. 5(a), used in a computation with a stream alignment conflict, such as `Z[i] = Y[i-1] + Y[i] + Y[i+1]`.

Fig. 5(b) shows the same set of data elements from a different logical view, as a two-dimensional 4×6 matrix. With row-major ordering used by C, such a 2D matrix will have exactly the same memory layout as the 1D matrix Y. Fig. 5(c) shows a 2D matrix that is the transpose of the 2D matrix in Fig. 5(b), i.e., a dimension-lifted transpose of the original 1D matrix in Fig. 5(a). Finally, Fig. 5(d) shows a 1D view of the 2D matrix in Fig. 5(c).

It can be seen that the data elements A and B, originally located in adjacent memory locations Y[0] and Y[1], are now spaced farther apart in memory, both being in column zero but in different rows of the transposed matrix shown in Fig. 5(c). Similarly, data elements G and H that were adjacent to each other in the original layout are now in the same column but different rows of the dimension-lifted transposed layout. The layout in Fig. 5(c) shows how elements would map to slots in vector registers that hold four elements each. The computation of A+B, G+H, M+N, and S+T can be performed using a vector operation after loading contiguous elements [A,G,M,S] and [B,H,N,T] into vector registers.

3.2 Stencil Computations on Transformed Layout

Fig. 6 provides greater detail on the computation using the transformed layout after a dimension-lifted transposition. Again, consider the following computation:

```
          for (i = 1; i < 23; ++i)
Sb:         Z[i] = Y[i-1] + Y[i] + Y[i+1];
```

Sixteen of the twenty two instances of statement **Sb** can be computed using full vector operations: [A,G,M,S] + [B,H,N,T] + [C,I,O,U]; [B,H,N,T] + [C,I,O,U] + [D,J,P,V]; [C,I,O,U] + [D,J,P,V] + [E,K,Q,W]; and [D,J,P,V] + [E,K,Q,W] + [F,L,R,X]. In Fig. 6, these fully vectorized computations are referred to as the "steady state". Six statement instances (computing E+F+G, F+G+H, K+L+M, L+M+N, Q+R+S, and R+S+T) represent "boundary" cases since the operand sets do not constitute full vectors of size four. One possibility is to perform the operations for these boundary cases as scalar operations,

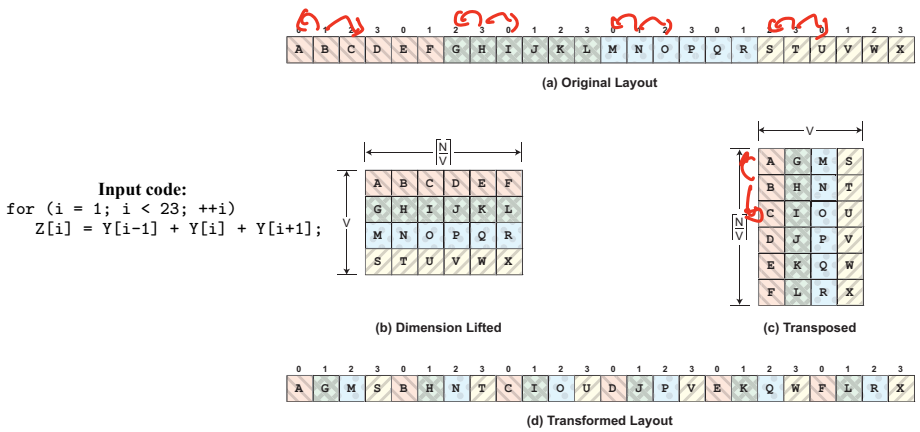


Fig. 5. Data layout transformation for SIMD vector length of 4

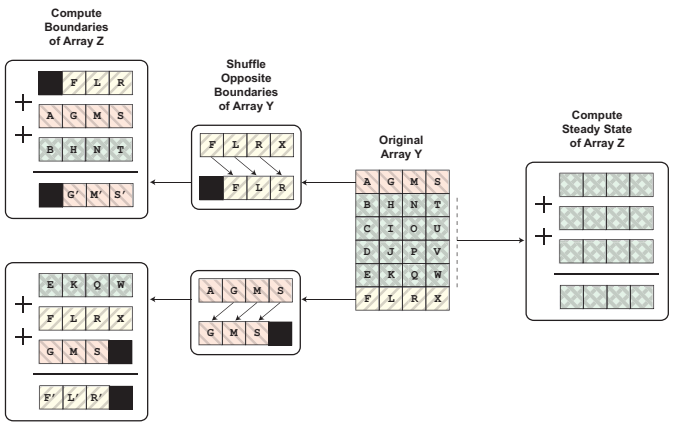


Fig. 6. Illustration of steady state and boundary computation

however, it is possible to use masked vector operations to perform them more efficiently. Fig. 6 illustrates the boundary cases corresponding to elements at the top and bottom rows in the transposed layout.

At the top boundary, a "ghost cell" is created by performing a right-shift to the bottom row vector [F,L,R,X] by one position to align [X,F,L,R] above [A,G,M,S]. The vector operation [X,F,L,R]+[A,G,M,S]+[B,H,N,T] is then performed and a masked write is performed so that only the last three components of the result vector get written. The bottom boundary is handled similarly, as illustrated in Fig. 6.

Higher order stencils simply result in a larger boundary area. Further, the dimension-lifted transpose layout transformation can be applied in the same manner for multi-dimensional arrays, as illustrated in Fig. 7. The fastest varying dimension (rightmost dimension for C arrays) of a multi-dimensional array is subjected to dimension-lifting and transposition (with padding if the size of that dimension is not a perfect multiple of

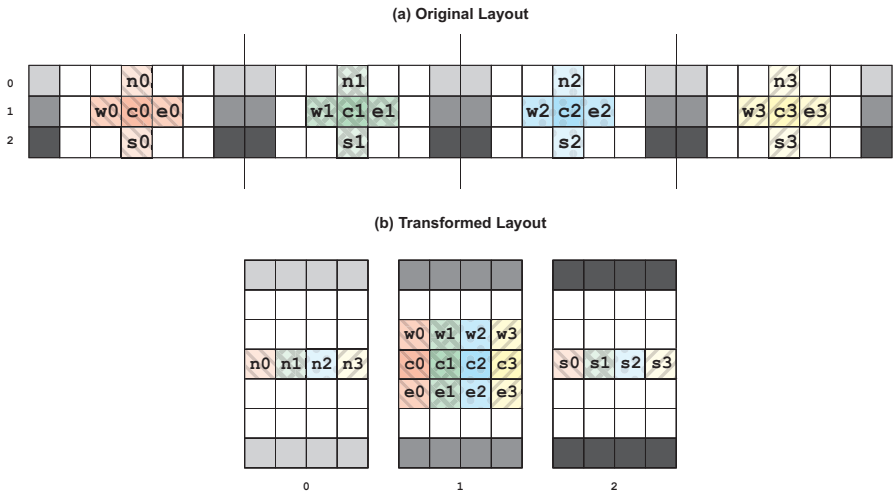


Fig. 7. Illustration for 2D 5-point stencil

vector length). Fig. 7 highlights a set of array locations accessed by a 2D 5-point stencil computation on a four wide vector architecture.

In the following section, we present a compiler framework to automatically detect where dimension lifting can be used to overcome the performance loss due to stream alignment conflicts.

4 Framework for Stream Alignment Conflict Detection

Stream alignment conflicts often occur when vectorizing stencil computations, as shown in the examples of Section 2. Our goal in this section is to develop compiler algorithms for detecting stream alignment conflicts. Note that stream alignment conflict is not limited to stencil computations; therefore, dimension-lifted transposition is useful for a more general class of programs. In this paper, we limit the discussion to vectorizable innermost loops. We assume that high-level transformations to expose parallel innermost loops that are good vectorization candidates have already been applied to the code [1]. The first task is to collect all vectorizable innermost loops. Some conditions must hold for our analysis to be performed: (1) all memory references are either scalars or accesses to (multidimensional) arrays; (2) all dependences have been identified; and (3) the innermost loop variable has a unit increment. Note that our framework handles unrolled loops as well; for ease of presentation, we will assume a non-unrolled representation of unrolled loops.

4.1 Program Representation

We first define our representation of candidate loops to be analyzed. The objective is to detect (a) if a loop is vectorizable, and (b) if it has a stream alignment conflict that can be resolved using dimension-lifted transposition of accessed arrays. We operate on an abstract program representation that is independent of the programming language and analyze the memory references of candidate innermost loops.

Innermost loops Innermost loops are the only candidates we consider for vectorization after dimension-lifted transposition of arrays. As stated earlier, we assume that such loops have a single induction variable that is incremented by one in each iteration of the loop. This is required to ensure the correctness of the analysis we use to characterize the data elements accessed by an innermost loop. We also require the innermost loops to have a single entry and exit point, so that we can precisely determine where to insert the data layout transformation (i.e., dimension-lifted transposition) code.

Memory references A loop contains a collection of memory references. All data elements accessed by any given iteration of the loop must be statically characterized by an access function f of the enclosing loop indices and other parameters. For a given execution of the loop whose iterator is `i`, all function parameters beside `i` must be constant. f can be multidimensional for the case of an array reference, with one dimension in f for each dimension of the array. For instance, for the reference `B[i][alpha+M*j]` with innermost loop iterator j, the access function is $f_B(j) = (i, alpha + M.j)$.

4.2 Candidate Vector Loops

Dimension-lifted transposition may be useful only for arrays referenced in vectorizable loops. We now provide a simple compiler test to determine which inner loops are candidates for vectorization, based on data dependence analysis of the program. Specifically, we require a candidate innermost loop to meet the following constraints[1]: (1) the loop does not have any loop-carried dependence; and (2) two consecutive iterations of the loop access either the same memory location or two locations that are contiguous in memory, i.e., the loop has either stride-zero or stride-one accesses.

Loop-carried dependences To determine if a loop is parallel, we rely on the standard concept of dependence distance vectors [19]. There is a dependence between two iterations of a loop if they both access the same memory location and one of these accesses is a write. If there is no loop-carried dependence, then the loop is parallel.

We now define the concept of a *array-distance vector* between two references, restricted to the inner loop (all other variables are assumed constant).

Definition 1 (Array-Distance Vector between two references). *Consider two access functions f_A^1 and f_A^2 to the same array `A` of dimension n. Let $\imath$ and $\imath'$ be two iterations of the innermost loop. The array-distance vector is defined as the n-dimensional vector* $\delta(\imath,\imath')_{f_A^1,f_A^2} = f_A^1(\imath) - f_A^2(\imath')$.

Let us illustrate Definition 1 with two references `A[i][j]` and `A[i-1][j+1]`, enclosed by loop indices i and j with j as the innermost loop iterator. We have $f_A^1(j) = (i, j)$ and $f_A^2(j) = (i-1, j+1)$. The array-distance vector is

$$\delta(j,j')_{f_A^1,f_A^2} = (1, j - j' - 1)$$

A necessary condition for the existence of a dependence carried by the innermost loop is stated below.

Definition 2 (Loop-carried dependence). *There exists a dependence carried by the innermost loop between two references f_A^1 and f_A^2, at least one of which is a write to memory, if there exists $\imath \neq \imath'$ such that* $\delta(\imath,\imath')_{f_A^1,f_A^2} = 0$.

[1] The case of vectorizing reductions is not addressed here.

Note that $\delta(\iota,\iota')_{f_A^1,f_A^2}$ requires the values of loop iterators other than the innermost loop to be fixed. Also, when $\delta(\iota,\iota')_{f_A^1,f_A^2}$, the difference between the access functions, still contains symbols beyond the inner-loop iterator (e.g., $\delta(\iota,\iota')_{f_A^1,f_A^2} = (k, alpha)$) we conservatively assume a loop-carried dependence unless the value of the symbols can be determined statically.

Stride-one memory accesses The second condition for a loop to be a SIMD-vectorization candidate is that all memory accesses are such that two consecutive iterations access either consecutive memory locations or an identical location, for all iterations of the innermost loop.

To test for this property, we operate individually on each access function. Memory references that always refer to the same location for all values of the innermost loop are discarded: they are the equivalent to referencing a scalar variable. For each of the remaining references, we form the array-distance vector between two consecutive iterations, and ensure the distance between the two locations is equal to 1. For the case of multidimensional arrays, for the distance in memory to be 1, the array-distance vector must have a zero value in all but the last dimension where it must be one. This is formalized as follows:

Definition 3 (Stride-one memory access for an access function). *Consider an access function f_A surrounded by an innermost loop. It has stride-one access if* $\forall \iota$, $\delta(\iota, \iota+1)_{f_A,f_A} = (0,\ldots,0,1)$.

For example, consider `A[i][j]` surrounded by the innermost j loop. $f_A = (i,j)$ and $\delta(j,j+1)_{f_A,f_A} = (0,1)$. Hence the j loop has stride-one access for f_A as above. Let us suppose that for the same f_A, loop i is the innermost loop; in this case, we would have $\delta(i,i+1)_{f_A,f_A} = (1,0)$ which does not satisfy the condition in Definition 3. Therefore loop i does not have stride-one access with respect to f_A.

4.3 Detection of Stream Alignment Conflict

A stream alignment conflict occurs when a given memory location is required to be mapped to two different vector slots. So, it follows that if a memory element is not used twice or more during the execution of the innermost loop, then a stream alignment conflict cannot occur. If some memory element is reused, one of two cases apply: either the element is reused only during the same iteration of the innermost loop, or it is used by another iteration of the innermost loop. Stream alignment conflict can occur only for the latter; if the element is reused during the same iteration, it is mapped to the same vector slot and there is no stream alignment conflict.

Cross-iteration reuse due to distinct references Our principle to detect if a given memory location is being referenced by two different iterations follows the concept of loop-carried dependence, but we extend now to *all pairs* of references and not limit to those containing a write operation.

For a given innermost loop, we thus compute the array-distance vector for all pairs of distinct references f^1 and f^2 to the same array, and see if there exists two distinct iterations such that

$$\delta(\iota,\iota')_{f^1,f^2} = 0$$

Consider the example of Figure 8(a). For the case of $f_B^1 = (i, j+1)$ and $f_B^2 = (i, j)$, we can determine if there is cross-iteration reuse by forming the constraint $\delta(j, j\prime)_{f_B^1, f_b^2} = (0, j+1-j\prime)$ and solving the system of constraints

$$\mathcal{S} : \begin{cases} j \neq j\prime \\ 0 = 0 \\ j+1-j\prime = 0 \end{cases}$$

Since S has a solution, there is cross-iteration reuse in the innermost loop. Technically, the presence of a solution to $\mathcal{S}$ is only a necessary condition for cross-iteration reuse. Some cases of cross-iteration reuse may not be detected, typically when distinct variables used in the access functions have the same value at runtime: the *evaluation* of the expression of δ may be 0, but simply doing the difference of the symbols may not give 0. Addressing this issue requires classical data-flow analysis problem for arrays [19] and can be solved with a more complete analysis and under stronger assumptions on the form of the input code. In our framework, the only consequence of not detecting dynamic cross-iteration reuse is that we may not have applied dimension-lifted transposition at places where it could have been useful. Our analysis is conservative in that sense but it requires only minimal assumptions on the input code.

```
for (i = lbi; i < ubi; ++i)
  for (j = lbj; j < ubj; ++j)
    A[i][j] = B[ i ][j+1] +
              B[ i ][ j ] +
              B[i-1][ j ];
```

(a)

```
for (i = lbi; i < ubi; ++i)
   for (j = lbj; j < ubj; ++j)
   {
R:   A[i][j] += B[i][j-1];
S:   C[i][j] += B[i][ j ];
   }
```

(b)

```
for (i = lbi; i < ubi; ++i)
{
  A[i][lbj] += B[i][lbj-1];
  for (j = lbj + 1; j < ubj; ++j)
  {
    A[i][ j ] += B[i][j-1];
    C[i][j-1] += B[i][j-1];
  }
  C[i][ubj] += B[i][ubj];
}
```

(c)

Fig. 8. Code examples

Stream offset The presence of a cross-iteration reuse does not necessarily imply the presence of a stream alignment conflict. In particular, all cases where cross-iteration reuse can be removed by iteration shifting (that is, a simple offset of the stream) do not correspond to a stream alignment conflict. Thus, in order to detect only the set of arrays to be dimension-lifted and transposed, we need to prune the set of references which do not generate a stream alignment conflict from the set of all references with cross-iteration reuse. .

Consider the example of Figure 8(b) that has cross-iteration reuse. It is possible to modify the innermost loop such that the reuse disappears via *iteration shifting* [19]. Shifting is the iteration space transformation view of manipulating the stream offset, as it influences which specific *instance* of the statements are being executed under the same iteration of loop j. Consider shifting the second statement by 1, the transformed code is shown in Figure 8(c).

Definition 4 (Stream offset via shifting). *Consider two statements R and S. Changing the stream offset is performed by shifting the iterations of R with respect to the iterations of S by a constant, scalar factor* σ_R*. All streams used by R have the same offset* σ_R*.*

There are two important properties of shifting for stream offset. First, as shown above, shifting can make cross-iteration reuse disappear in some cases. It can also introduce new cross-iteration reuse by assigning different offsets to streams associated with the same array.

Second, shifting is not always a legal reordering transformation. It may reorder the statements inside the loop, and change the semantics. This translates to a strong condition on the legality of iteration shifting.

Definition 5 (Legality of iteration shifting). *Consider a pair of statements R and S such that they are surrounded by a common vectorizable innermost loop. Let* σ_R *be (resp.* σ_S*) the shift factor used to offset the streams of R (resp. S). If there is a dependence between R and S, then to preserve the semantics it is sufficient to set* $\sigma_R = \sigma_S$.

To determine if iteration shifting can remove cross-iteration reuse, we first observe that it changes the access functions of a statement R by substituting j with $j - \sigma_R$, given j as the innermost loop iterator. The variable j can be used only in the last dimension of the access function, since the loop is vectorizable with stride-one access. We then formulate a problem similar to that of cross-iteration reuse analysis, with the important difference being that we seek values of σ that make the problem infeasible; indeed if the problem has no solution, then there is no cross-iteration reuse. We also restrict it to the pairs of references such that all but the last dimension of the access functions are equal, as all cases of reuse require this property. To ease the solution process, we thus reformulate it into a feasibility problem by looking for a solution where $\imath = \imath'$ that is independent of the value of $\imath$.

Returning to the example in Figure 8(b), we have for B, $f_B^1(j) = (i, j-1)$ and $f_B^2(j) = (i, j)$. We first integrate the offset factors σ into the access function. As the two statements are not dependent, we have one factor per statement that can be independently computed. The access functions to consider are now $f_B^1(j) = (i, j - \sigma_R - 1)$ and $f_B^2(j) = (i, j - \sigma_S)$. We consider the problem

$$\mathcal{T} : \begin{cases} j = j' \\ j - \sigma_R - 1 - j' + \sigma_S = 0 \end{cases}$$

If $\mathcal{T}$ has a solution for all values of j, that is, a solution independent of j then there is no cross-iteration reuse. For $\mathcal{T}$, $\sigma_R = 0$ and $\sigma_S = 1$ is a valid solution and this iteration shifting removes all cross-iteration reuse.

In order to find a valid solution for the whole inner-loop, it is necessary to combine all reuse equalities in a single problem: the σ variables are shared for multiple references and must have a unique value. Hence, for code in Figure 8(a), the full system to solve integrates $\delta(j, j')_{f_A^1, f_A^2}$, is augmented with σ_R and σ_S, and is shown below.

$$\mathcal{T} : \begin{cases} j = j' & \\ j - \sigma_R - 1 - j' + \sigma_S = 0 & \text{Conditions for B} \\ j - \sigma_R - j' - \sigma_S = 0 & \text{Conditions for A} \end{cases}$$

$\mathcal{T}$ has no solution, showing that there is no possible iteration shifting that can remove all cross-iteration reuse in Figure 8(a). Dimension-lifted transposition is thus required.

Putting it all together We now address the general problem of determining if a given innermost loop suffers from a stream alignment conflict that can be solved via dimension-lifted transposition. That is, we look for cross-iteration reuse that cannot be eliminated via iteration shifting.

First, let us step back and precisely define in which cases dimension-lifted transposition can be used to solve a stream alignment conflict. Dimension-lifted transposition in essence spreads out the memory locations of references involved in a stream alignment conflict. In order to ensure there is no conflict remaining, one must precisely know, at compile-time, the distance in memory required to separate all elements. This distance must be a constant along the execution of the innermost loop. This translates to an additional constraint on the cross-iteration reuse that is responsible for the stream alignment conflict: the reuse distance must be a constant. We define the scope of applicability of dimension-lifted transposition as follows.

Definition 6 (Applicability of dimension-lifted transposition). *Consider a collection of statements surrounded by a common vectorizable inner loop. If there exists cross-iteration reuse of a constant distance that cannot be eliminated by iteration shifting, then the stream alignment conflict can be solved with dimension-lifted transposition.*

If an array accessed in a candidate vector inner loop is dimension-lifted-and-transposed, all arrays in the inner loop are also dimension-lifted-and-transposed. The data layout transformation implies significant changes in the loop control and in the order the data elements are being accessed. All arrays must be dimension-lifted unless some computations simply could not be vectorized anymore. We present in Figure 9 a complete algorithm to detect which arrays are to be transformed by dimension-lifted transposition in a program.

```
Input P: input program
Output Arrays: the set of arrays to be dimension-lifted

ℒ ← ∅
forall innermost loops l in P do
  forall arrays A referenced in l do
    /* Check loop-carried dependence */
    forall write references f_A^1 to A do
      forall references f_A^2 to A do
        S ← { ∃ (ι,ι'), δ(ι,ι')_{f_A^1,f_A^2} = 0 }
        if S ≠ ∅ then goto next loop l
    /* Check stride-one */
    forall references f_A to A do
      S ← { ∀ ι, δ(ι,ι+1)_{f_A,f_A} = (0,...,0,1) }
      if S = ∅ then goto next loop l
    /* Check scalar reuse distance */
    forall pairs of references f_A^1, f_A^2 to A do
      S ← {∃ (ι,ι') δ(ι,ι')_{f_A^1,f_A^2} = 0 }
      if S ≠ ∅ then
        S ← { α ∈ ℤ, δ(ι,ι)_{f_A^1,f_A^2} = (0,...,0,α) }
        if S = ∅ then goto next loop l
  ℒ ← ℒ ∪ l
forall l ∈ ℒ do
  /* Check conflict after iteration shifting */
  T ← createIterationShiftingProblem(l)
  if T = ∅ then
    Arrays(l) ← all arrays in l
```

Fig. 9. Algorithm to detect arrays to be dimension-lifted-and-transposed

Procedure `createIterationShiftingProblem` creates a system of equalities that integrates the shift factors σ as shown in Section 4.3. If this system has no solution, then at least one cross-iteration reuse remains even after iteration shifting. Since we have

prevented the cases where the reuse is not a scalar constant, then the conflict can be solved with dimension lifting. We thus place all arrays of the loop into the list of arrays to be dimension-lifted-and-transposed.

While solving $\mathcal{T}$, we compute values for σ to remove cross-iteration reuse. When it is not possible to remove all cross-iteration reuses with iteration shifting, we compute values for σ that minimize the distance between two iterations reusing the same element. The largest among all reuse distances in the iteration-shifted program is kept and used during code generation to determine the boundary conditions.

5 Experimental Evaluation

The effectiveness of the dimension-lifting transformation was experimentally evaluated on several hardware platforms using stencil kernels from a variety of application domains. First, we describe the hardware and compiler infrastructure used for experiments. Next, the stencil kernels used in the experiments are described. Finally, experimental results are presented and analyzed.

5.1 Hardware

We performed experiments on three hardware platforms: AMD Phenom 9850BE, Intel Core 2 Quad Q6600, and Intel Core i7-920. Although all are x86 architectures, as explained below, there are significant differences in performance characteristics for execution of various vector movement and reordering instructions.

Phenom 9850BE The AMD Phenom 9850BE (*K10h* microarchitecture) is an x86-64 chip clocked at 2.5 GHz. It uses a 128b FP add and 128b FP multiply SIMD units to execute a maximum of 8 single precision FP ops per cycle (20 Gflop/s). The same SIMD units are also used for double precision operations, giving a peak throughput of 10 Gflop/s. Unaligned loads are penalized on this architecture, resulting in half the throughput of aligned loads and an extra cycle of latency. The SSE shuffle instruction `shufps` is used by ICC for single precision inter- and intra-register movement. Double precision stream alignment conflicts are resolved by ICC generating consecutive `movsd` and movhpd SSE instructions to load the low and high elements of a vector register.

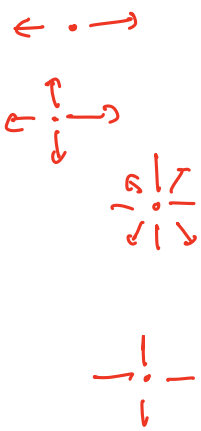
Core 2 Quad Q6600 The Intel Core 2 Quad Q6600 (*Kentsfield* microarchitecture) is an x86-64 chip running at 2.4 GHz. Like the Phenom, it can issue instructions to two 128-bit add and multiply SIMD units per cycle to compute at a maximum rate of 19.2 single precision GFlop/s (9.6 double precision Gflop/s). The `movups` and `movupd` unaligned load instructions are heavily penalized on this architecture. Aligned load throughput is 1 load/cycle. Unaligned load throughput drops to 5% of peak when the load splits a cache line and 50% of peak in all other cases. ICC generates the `palignr` SSSE3 instruction for single precision inter- and intra-register movement on Core 2 Quad. Double precision shifts are accomplished with consecutive `movsd-movhpd` sequences as previously described.

Core i7-920 The Intel Core i7-920 (*Nehalem* microarchitecture) is an x86-64 chip running at 2.66 GHz. SIMD execution units are configured in the same manner as the previously described x86-64 processors, leading to peak FP throughput of 21.28 single precision GFlop/s and 10.64 double precision Gflop/s. Unaligned loads on this processor

are very efficient. Throughput is equal to that of aligned loads at 1 load/cycle in all cases except cache line splits, where it drops to 1 load per 4.5 cycles. Single precision code generated by ICC auto-vectorization uses unaligned loads exclusively to resolve stream alignment conflicts. Double precision code contains a combination of consecutive `movsd-movhd` sequences and unaligned loads.

5.2 Stencil Codes

We evaluated the use of the dimension-lifting layout transformation on seven stencil benchmarks, briefly described below.



Jacobi 1/2/3D The Jacobi stencil is a symmetric stencil that occurs frequently both in image processing applications as well as with explicit time-stepping schemes in PDE solvers. We experimented with one-dimensional, 2D, and 3D variants of the Jacobi stencil, and used the same weight for all neighbor points on the stencil and the central point.

In the table of performance data below, the 1D Jacobi variant is referred as J-1D. For the 2D Jacobi stencil, both a five point "star" stencil (J-2D-5pt) and 9 point "box"(J-2D-9pt) stencil were evaluated A seven point "star" stencil (J-3D-9pt) was used to evaluate performance of Jacobi 3D code.

Heattut 3D This is a kernel from the Berkeley stencil probe and is based on a discretization of the heat equation PDE.[17].

FDTD 2D This kernel is the core computation in the widely used Finite Difference Time Domain method in Computational Electromagnetics [34]

Rician Denoise 2D This application performs noise removal from MRI images and involves an iterative loop that performs a sequence of stencil operations.

Problem Sizes We assume the original program is tiled such that the footprint of a tile does not exceed the L1 cache size, thus all arrays are sized to fit in the L1 data cache. As is common for stencil codes, for each of the benchmarks, there is an outer loop around the stencil loops, so that any one-time layout transformation cost to copy from an original standard array representation to the transformed representation involves a negligible overhead.

Code versions For each code, three versions were tested:

- Reference, compiler auto-vectorized
- Layout transformed, compiler auto-vectorized
- Layout transformed, explicitly vectorized with intrinsics

Vector intrinsic code generation Vector intrinsic code generation is based on the process shown in Figure 6. An outline of the steps in code generation is provided next.

Convert stencil statement(s) into intrinsic equivalents We convert C statements into vector intrinsic equivalents. For example, consider the following 3 point 1D Jacobi statement:

```
a[i] = b[i-1] + b[i] + b[i+1];
```

This statement can be expressed in SSE intrinsics:

```
adlt[i] = _mm_add_ps(_mm_add_ps(bdlt[i-1],bdlt[i), bdlt[i+1]);
```

Note that `dlt` suffixed arrays have been layout transformed.

Generate boundary code The reuse distance information obtained with the framework of Section 4 above is used to generate boundary code from the intrinsic statements. This code contains the appropriate shifts and masked stores required to maintain program correctness.

Generate intrinsic steady state code Again, reuse distance information is used to generate a vector intrinsic inner loop. This loop, along with boundary code, replaces the original inner loop. Finally, well-known loop unrolling and register blocking optimizations are performed. It is interesting to note that unrolling the vanilla C versions of the codes did not improve performance (in many cases impacted performance negatively), while unrolled versions of the vector intrinsic code resulted in performance improvement.

5.3 Results

Absolute performance and relative improvement for single and double precision experiments across all platforms and codes are given in Figure 10. Intel C Compiler icc v11.1 with the '`-fast`' option was used for all machines.Vectorization pragmas were added to the inner loops of reference and layout transformed codes to force ICC auto-vectorization.

		Phenom				Core2 Quad				Core i7			
		SP		DP		SP		DP		SP		DP	
		GF/s	Imp.	GF/s	Imp.	GF/s	Imp.	GF/s	Imp.	GF/s	Imp.	GF/s	Imp.
J-1D	Ref.	4.27	1.00×	3.08	1.00×	3.71	1.00×	2.46	1.00×	8.67	1.00×	3.86	1.00×
	DLT	7.68	1.80×	3.79	1.23×	9.42	2.54×	2.83	1.15×	10.55	1.22×	4.01	1.04×
	DLTi	11.38	2.67×	5.71	1.85×	13.95	3.76×	7.01	2.85×	15.35	1.77×	7.57	1.96×
J-2D-5pt	Ref.	6.96	1.00×	2.71	1.00×	3.33	1.00×	2.94	1.00×	8.98	1.00×	4.54	1.00×
	DLT	9.00	1.29×	3.75	1.38×	8.86	2.66×	4.58	1.56×	10.20	1.14×	5.18	1.14×
	DLTi	11.31	1.63×	5.67	2.09×	11.58	3.48×	5.85	1.99×	13.12	1.46×	6.58	1.45×
J-2D-9pt	Ref.	4.48	1.00×	3.21	1.00×	4.21	1.00×	2.72	1.00×	8.30	1.00×	4.11	1.00×
	DLT	7.71	1.72×	3.81	1.18×	8.04	1.91×	4.08	1.50×	10.23	1.23×	5.23	1.27×
	DLTi	12.26	2.74×	6.11	1.90×	12.01	2.85×	6.03	2.22×	13.62	1.64×	6.80	1.65×
J-3D	Ref.	6.01	1.00×	2.90	1.00×	6.07	1.00×	3.04	1.00×	9.04	1.00×	4.64	1.00×
	DLT	6.84	1.14×	3.73	1.29×	8.07	1.33×	4.25	1.40×	9.46	1.05×	5.02	1.08×
	DLTi	10.08	1.68×	5.36	1.85×	10.36	1.71×	5.31	1.75×	12.02	1.33×	6.04	1.30×
Heatttut-3D	Ref.	6.06	1.00×	3.02	1.00×	6.64	1.00×	3.29	1.00×	8.75	1.00×	4.55	1.00×
	DLT	7.12	1.18×	3.36	1.11×	8.71	1.31×	4.45	1.35×	9.99	1.14×	4.91	1.08×
	DLTi	9.59	1.58×	5.12	1.70×	8.86	1.33×	4.45	1.35×	11.99	1.37×	6.05	1.33×
FDTD-2D	Ref.	5.86	1.00×	3.26	1.00×	6.42	1.00×	3.35	1.00×	8.72	1.00×	4.34	1.00×
	DLT	6.89	1.18×	3.65	1.12×	7.71	1.20×	4.03	1.20×	8.91	1.02×	4.73	1.09×
	DLTi	6.64	1.13×	3.41	1.05×	8.03	1.25×	4.03	1.20×	9.74	1.12×	4.82	1.11×
Rician-2D	Ref.	3.29	1.00×	1.93	1.00×	1.87	1.00×	1.27	1.00×	3.98	1.00×	2.16	1.00×
	DLT	3.46	1.05×	2.40	1.25×	2.59	1.39×	1.27	1.00×	4.13	1.04×	2.23	1.03×
	DLTi	8.09	2.46×	2.56	1.33×	8.50	4.55×	1.27	1.00×	11.31	2.84×	2.23	1.03×

Fig. 10. Summary of experimental results. Ref is the unoptimized, auto-vectorized version. DLT is the layout transformed, auto-vectorized version. DLTi is the layout transformed version implemented with vector intrinsics.

Double Precision Double precision results are shown in columns labeled DP of Figure 10. Significant performance gains are achieved across all platforms and on all benchmarks.

ICC auto-vectorized DLT code equaled or improved upon reference code performance in all cases. The harmonic means of relative improvements across all double precision benchmarks on x86-64 were 1.10× (Core i7), 1.22× (Phenom), and 1.28× (Core

at runtime. → This is what the CUDA/OpenCL code does.

2 Quad). Individual benchmark improvements range from, worst case, 1.00× (2D Rician Denoise on Core 2 Quad) to a best case of 1.56× (5 point 2D Jacobi on Core 2 Quad).

The auto-vectorized layout transformed code was fast but certain areas of it were still very inefficient. While ICC automatically unrolled the inner loop of reference code, no such unrolling was done for the layout transformed code. Further, ICC generated long sequences of scalar code for boundary computations.

These deficiencies were addressed in the intrinsic versions of the codes. Scalar boundary code was replaced with much more efficient vector code, and all inner loops were unrolled. Further gains can be also be attributed to register blocking and computation reordering.

Intrinsic codes equaled or improved upon auto-vectorized versions in all cases, with a worst case improvement equal to reference (2D Rician Denoise on Core 2 Quad) and best case of 2.85× (Jacobi 1D on Core 2 Quad). Harmonic means of improvements over reference were 1.35× (Core i7), 1.60× (Phenom), and 1.57× (Core 2 Quad).

Why did we have to hand-vectorize?

Single Precision While most scientific and engineering codes use double precision for their computations, several image processing stencils use single precision. With the current SSE vector ISA, since only two double precision elements can fit in a vector, acceleration of performance through vectorization is much less than with single precision. However, the increasing vector size of emerging vector ISAs such as AVX and LRBni, imply that the performance improvement currently possible with single precision SSE will be similar to what we can expect for double precision AVX, etc. For these reasons we include single precision performance data for all benchmarks.

Significant single precision performance gains are achieved across all platforms and on all stencils. They are reported in Figure 10 under the SP columns.

Layout transformed code auto-vectorized by ICC ran significantly faster than reference code on all platforms. The harmonic means of relative performance improvements across all benchmarks on x86-64 were 1.11× (Core i7), 1.29× (Phenom), and 1.61× (Core 2 Quad). Individual benchmark improvements range from, worst case, 1.02× (2D FDTD on Core i7) to a best case of 2.66× (5 point 2D Jacobi on Core 2 Quad).

Vector intrinsic code optimizations again further increased the performance gains seen for auto-vectorized layout transformed code. All intrinsic codes were substantially faster than their corresponding auto-vectorized versions. Minimum relative improvement over reference on x86-64 was 1.12× (2D FDTD on Core i7) while maximum relative improvement was 4.55× (2D Rician Denoise on Core 2 Quad). Harmonic means of improvements over reference were 1.53× (Core i7), 1.81× (Phenom), and 2.15× (Core 2 Quad).

Discussion Performance gains for all x86-64 codes can be attributed to the elimination of costly intra-register movement, shuffle, and unaligned load instructions from inner loop code.

The performance gains on Core i7, while significant, were consistently the smallest of any platform tested. This is partly explained by the relatively small performance penalty associated with unaligned loads and shuffle on this CPU. Still, the DLT intrinsic versions achieve a 1.53× average performance improvement for single precision and 1.35× for double precision codes on this platform. In contrast, the *Kentsfield* Core 2 Quad, demonstrates consistently large performance improvements from layout transformation. This can mainly be attributed to poorly performing vector shuffle hardware.

Generally speaking, 1D Jacobi showed both the largest performance gains, and the fastest absolute performance, while higher dimensional stencils showed smaller, but still

significant improvement. Higher dimensional stencils have more operands and more intra-stencil dependences. This leads to higher register occupancy, higher load / store unit utilization, and more pipeline hazards / stalls for these codes. This combination of factors leads less improvement with respect to the 1D case. General and application-specific optimizations based on the data layout transformation described in this work could likely achieve higher performance through careful instruction scheduling and tuning of register block sizes to address these issues.

6 Related Work

A number of works have addressed optimizations of stencil computations on emerging multicore platforms [7], [16], [17], [6], [27], [26], [11], [37], [10], [4], [9], [40], [38], [41], [8], [39]. In addition, other transformations such as tiling of stencil computations for multicore architectures have been addressed in [43], [25], [21], [34]. Recently, memory customization for stencils has been proposed in [36].

Automatic vectorization has been the subject of extensive study in the literature [19, 42]. There has been significant recent work in generating effectice code for SIMD vector instruction sets in the presence of hardware alignment and stride constraints as described in [12, 44, 45, 31, 13]. The difficulties of optimizing for a wide range of SIMD vector architectures are discussed in [29, 14]. In addition, several other works have addressed the exploitation of SIMD instruction sets [22, 24, 23, 30, 32, 31, 28]. All of these works only address SIMD hardware alignment issues. The issues of algorithmic stream alignment addressed in this paper are distinctly different from the problem addressed in those works and the dimension-lifted transposition solution that we have developed has a significant impact on performance even on SIMD architectures where hardware misalignment does not significantly degrade performance.

Stream alignment shares a lot similarties with array alignment in data-parallel languages [2, 5, 20] and several related works. None of these works, however, considered dimension-lifted transposition of accessed arrays. There has been prior work attempting to use static linear data layout optimizations (such as permutations of array dimensions) to improve spatial locality in programs [33, 18]. These works do not address dimension-lifted transposition. Rivera and Tseng [35] presented data padding techniques to avoid conflict misses. Recently, linear data layout transformations to improve vector performance have been proposed [15].

To avoid conflict misses and false sharing, Amarasinghe's work [3] maps data accessed by a processor to contiguous memory locations by using strip-mining and permutation of data arrays. In contrast, our approach attempts remap data in order to spread out reuse carrying data in the innermost loops in order to have them map to the same vector register slot; this avoids alignment conflicts and eliminates the need for extra loads or inter- and intra-register data movement.

7 Conclusions

This paper identifies, formalizes and provides an effective solution for a fundamental problem with optimized implementation of stencil computations on short-vector SIMD architectures. The issue of stream alignment conflicts was formalized and a static analysis framework was developed to identify it. A novel nonlinear data layout transformation was proposed to overcome stream alignment conflicts. Experimental results on multiple targets demonstrate the effectiveness of the approach on a number of stencil kernels.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation through awards 0926127, 0926687, and 0926688, and by the U.S. Army through contract W911NF-10-1-0004. We thank the reviewers, especially "Reviewer 3", for constructive feedback that helped improve the paper.

References

1. R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM TOPLAS*, 9(4):491–542, 1987.
2. S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *PLDI*, pages 126–138, 1993.
3. J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In *PPoPP*, pages 166–178, 1995.
4. W. Augustin, V. Heuveline, and J. Weiss. Optimized stencil computation using in-place calculation on modern multicore systems. In *Euro-Par '09*, pages 772–784, 2009.
5. S. Chatterjee, J. Gilbert, R. Schreiber, and S. Teng. Automatic array alignment in data-parallel programs. In *POPL*, pages 16–28, 1993.
6. K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159, 2009.
7. K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08*, pages 1–12, 2008.
8. K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Auto-tuning the 27-point stencil for multicore. In *iWAPT2009*, 2009.
9. R. de la Cruz, M. Araya-Polo, and J. M. Cela. Introducing the *semi-stencil* algorithm. In *PPAM (1)*, pages 496–506, 2009.
10. H. Dursun, K. Nomura, W. Wang, M. Kunaseth, L. Peng, R. Seymour, R. Kalia, A. Nakano, and P. Vashishta. In-core optimization of high-order stencil computations. In *PDPTA*, pages 533–538, 2009.
11. H. Dursun, K.-I. Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta. A multilevel parallelization framework for high-order stencil computations. In *Euro-Par '09*, pages 642–653, 2009.
12. A. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *PLDI*, pages 82–93, 2004.
13. L. Fireman, E. Petrank, and A. Zaks. New algorithms for simd alignment. In *CC*, pages 1–15, 2007.
14. M. Hohenauer, F. Engel, R. Leupers, G. Ascheid, and H. Meyr. A simd optimization framework for retargetable compilers. *ACM TACO*, 6(1):1–27, 2009.
15. B. Jang, P. Mistry, D. Schaa, R. Dominguez, and D. R. Kaeli. Data transformations enabling loop vectorization on multithreaded data parallel architectures. In *PPOPP*, pages 353–354, 2010.
16. S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC '06*, pages 51–60, 2006.
17. S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *MSP '05*, pages 36–43, 2005.
18. M. Kandemir, A. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A linear algebra framework for automatic determination of optimal data layouts. *IEEE TPDS*, 10(2):115–135, 1999.
19. K. Kennedy and J. Allen. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann, 2002.
20. K. Kennedy and U. Kremer. Automatic data layout for distributed-memory machines. *ACM TOPLAS*, 20(4):869–916, 1998.

21. S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI*, pages 235–244, 2007.
22. S. Larsen and S. P. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*, pages 145–156, 2000.
23. S. Larsen, R. M. Rabbah, and S. P. Amarasinghe. Exploiting vector parallelism in software pipelined loops. In *MICRO*, pages 119–129, 2005.
24. S. Larsen, E. Witchel, and S. P. Amarasinghe. Increasing and detecting memory address congruence. In *IEEE PACT*, pages 18–29, 2002.
25. Z. Li and Y. Song. Automatic tiling of iterative stencil loops. *ACM TOPLAS*, 26(6):975–1028, 2004.
26. J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *ICS*, pages 256–265, 2009.
27. P. Micikevicius. 3d finite difference computation on gpus using cuda. In *GPGPU-2*, pages 79–84, 2009.
28. D. Naishlos, M. Biberstein, S. Ben-David, and A. Zaks. Vectorizing for a SIMdD DSP architecture. In *CASES*, pages 2–11, 2003.
29. D. Nuzman and R. Henderson. Multi-platform auto-vectorization. In *CGO*, pages 281–294, 2006.
30. D. Nuzman, M. Namolaru, A. Zaks, and J. H. Derby. Compiling for an indirect vector register architecture. In *CF*, pages 199–208, 2008.
31. D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In *PLDI*, pages 132–143, 2006.
32. D. Nuzman and A. Zaks. Outer-loop vectorization: revisited for short simd architectures. In *PACT*, pages 2–11, 2008.
33. M. O'Boyle and P. Knijnenburg. Nonsingular data transformations: Definition, validity, and applications. *IJPP*, 27(3):131–159, 1999.
34. D. Orozco and G. R. Gao. Mapping the FDTD Application to Many-Core Chip Architectures. In *ICPP*, pages 309–316, 2009.
35. G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *PLDI*, 1998.
36. M. Shafiq, M. Pericas, R. de la Cruz, M. Araya-Polo, N. Navarro, and E. Ayguade. Exploiting memory customization in fpga for 3d stencil computations. In *FPT*, pages 38–45, 2009.
37. A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.
38. J. Treibig, G. Wellein, and G. Hager. Efficient multicore-aware parallelization strategies for iterative stencil computations. *CoRR*, abs/1004.1741, 2010.
39. S. Venkatasubramanian and R. Vuduc. Tuned and wildly asynchronous stencil kernels for hybrid cpu/gpu systems. In *ICS*, pages 244–255, 2009.
40. G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. In *COMPSAC*, pages 579–586, 2009.
41. M. Wittmann, G. Hager, J. Treibig, and G. Wellein. Leveraging shared caches for parallel temporal blocking of stencil codes on multicore processors and clusters. *CoRR*, abs/1006.3148, 2010.
42. M. J. Wolfe. *High Performance Compilers For Parallel Computing*. Addison-Wesley, 1996.
43. D. Wonnacott. Achieving scalable locality with time skewing. *IJPP*, 30(3):1–221, 2002.
44. P. Wu, A. E. Eichenberger, and A. Wang. Efficient SIMD Code Generation for Runtime Alignment and Length Conversion. In *CGO*, pages 153–164, 2005.
45. P. Wu, A. E. Eichenberger, A. Wang, and P. Zhao. An integrated simdization framework using virtual vectors. In *ICS*, pages 169–178, 2005.