Automatic Vectorization of Interleaved Data Revisited

ANDREW ANDERSON, Lero, Trinity College Dublin
AVINASH MALIK, University of Auckland
DAVID GREGG, Lero, Trinity College Dublin

Automatically exploiting short vector instructions sets (SSE, AVX, NEON) is a critically important task for optimizing compilers. Vector instructions typically work best on data that is contiguous in memory, and operating on non-contiguous data requires additional work to gather and scatter the data. There are several varieties of non-contiguous access, including interleaved data access. An existing approach used by GCC generates extremely efficient code for loops with power-of-two interleaving factors (strides). In this paper we propose a generalization of this approach that produces similar code for any compile-time constant interleaving factor. In addition, we propose several novel program transformations which were made possible by our generalized representation of the problem. Experiments show that our approach achieves significant speedups for both power-of-two and non-power-of-two interleaving factors. Our vectorization approach results in mean speedups over scalar code of 1.77x on Intel SSE and 2.53x on Intel AVX2 in real-world benchmarking on a selection of BLAS Level 1 routines. On the same benchmark programs, GCC 5.0 achieves mean improvements of 1.43x on Intel SSE and 1.30x on Intel AVX2. In synthetic benchmarking on Intel SSE, our maximum improvement on data movement is over 4x for gathering operations and over 6x for scattering operations versus scalar code.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors — *Optimization*

General Terms: Algorithms, Performance, Design

Additional Key Words and Phrases: Vectorization, Interleaving, SIMD, Data Permutation

ACM Reference Format:
Andrew Anderson, Avinash Malik, and David Gregg, 2015. Automatic Vectorization of Interleaved Data Revisited. *ACM Trans. Architec. Code Optim.* V, N, Article A (January YYYY), 24 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Since the introduction of MAX-1 vector instructions into the PA-RISC processor family [Lee 1995], hardware support for short vector parallelism has become widespread. General-purpose processors (e.g. Intel SSE and AVX), embedded processors (ARM NEON, Movidius Myriad) and accelerators (Intel Xeon Phi) all provide significant short vector compute resources. Vector instruction execution offers parallel speedups without the complexity of executing multiple separate instructions in parallel, and is therefore usually efficient in the use of hardware resources and energy.

Ideally a compiler will automatically discover and exploit opportunities to use the vector units. However, automatic vectorization presents many problems in dependence analysis, code restructuring and data access patterns [Wolfe 1996]. A common data access pattern is where data items that are processed in different ways are inter-

This work was supported, in part, by Science Foundation Ireland grants 12/IA/1381 and 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

leaved together in memory. Unlike traditional vector processors which provided rich data access instructions for gather, scatter and strided memory operations, modern processors with vector extensions provide poor support for non-consecutive memory access[1]. On encountering interleaved memory access, the compiler must generate fast instruction sequences to pack and unpack non-consecutive data to and from vector registers for vectorization to be successful. Investigations of bottlenecks in SIMD programs have identified non-unit-stride memory access patterns as a particular concern [Talla et al. 2003; Maleki et al. 2011; Schaub et al. 2015].

Nuzman et al. [2006] proposed an auto-vectorization algorithm for interleaved data access patterns where the stride is a power-of-two. Given a loop with such an access pattern, the algorithm generates extremely efficient vectorized code by directly exploiting the structure of the access pattern. This approach is highly successful in practice, and is the technique of choice for loops with interleaved data access in the GCC compiler.

In this article we generalize this approach to handle arbitrary constant strides. Non-power-of-two strides are common in many real-world programs, so being able to deal with any compile-time constant stride allows the approach to be applied to a much wider class of loops. We first create a simple canonical vectorization of loops with interleaved access using vector permutation and blending (Section 2). Our algorithm then applies novel optimizations to reduce the number of permutation operations and merge blend instructions where possible (Section 3). Finally, we present an experimental evaluation of our approach for generated code with various strides and data types, and for loop kernels from real-world applications (Section 4.4). Several vectorization approaches have previously been proposed that can deal with specific cases of interleaved access by accident or design. In Section 5 we briefly discuss these approaches, and present the case for our approach which generates highly efficient vector code for loops with arbitrary constant interleaving patterns. Algorithms referred to in the text are collected in Appendix A.

2. TECHNIQUE

Strided memory accesses can be represented by a function which translates a loop iteration variable so that it selects elements of an array.

DEFINITION 1 (ACCESS). *We represent a strided access with iteration variable i as a function a of the form*

$$a(i) = b + u * (\texttt{stride} * i + \texttt{offset})$$

where b is the base address of the array in bytes, and u the unit size in bytes of an array element. The access is consecutive when $|\texttt{stride}|$ is 1, and nonconsecutive otherwise. Note that the sign of the stride or offset may be negative.

Our approach to vectorizing such accesses follows a simple, general scheme: we cover the memory range accessed with non-overlapping vector loads or stores, and map individual accessed elements to loaded or stored lanes. The problem of vectorizing a strided access then becomes the problem of composing an ordered subset of loaded or stored lanes to or from a single vector register. In this article, we develop our approach by first showing how to create a vectorized code sequence in a simple, canonical form, and by application of successive optimizations, refine it into the final form which will be emitted.

[1]Note that Intel's AVX2 vector instruction set includes gather instructions for non-contiguous data loading. However, we found that the performance of these instructions was poor on our experimental platform.

While our approach does not require aligned access, it may be desirable for performance reasons. Any implementation may apply a wide variety of possible techniques to ensure aligned access [Eichenberger et al. 2004; Fireman et al. 2007]. One approach is to load extra lanes and discard those unused, treating the vector register file as a compiler-controlled cache [Shin et al. 2002]. However, this tactic has some corner cases: when the base address of the array is misaligned or the array does not contain enough data, an implementation using this tactic may have to apply array padding, or rely on masked vector loads or stores. Similarly, when loop trip counts are not a multiple of the vectorization factor (VF), extra iterations may need to be peeled and performed as scalar iterations.

2.1. Notation and Presentation

We specify our code generation in terms of the following abstract instruction set. There are several motivations for this choice of instructions. Many architectures provide a large number of distinct specialized data reorganization instructions. Were we to state our code generation in terms of specialized instructions, it would restrict the applicability of our techniques to architectures with support for those instructions. The choice of a simple, generic form of permute and blend instructions ensures our approach is more widely applicable.

Second, the program transformations we propose in this article are quite succinct when expressed in terms of these simple instructions. Including many specialized data reorganization instructions would significantly complicate the presentation of our techniques.

Finally, it is important to note that on architectures which do have highly specialized data reorganization instructions, they do not go unused. Many multimedia architectures such as Intel SSE, AVX, AVX2, and ARM NEON provide such instructions in addition to instructions corresponding to those we include. However, many of the highly specialized native instructions for data reorganization can be expressed in terms of a short sequence of more generic permutes and blends. In this scenario, traditional tree-parsing instruction selection techniques [Aho et al. 1989] are very effective at selecting highly-specialized native instructions to cover the sequences of simplified operations which we generate. We detail our approach to native code generation in Section 4.3.

Instruction	Arguments
load	(Vector target, Pointer source)
store	(Vector source, Pointer target)
permute	(Vector source, Vector mask, Vector target)
blend	(Vector left, Vector right, Vector mask, Vector target)

Fig. 1: The target intermediate representation for code generation.

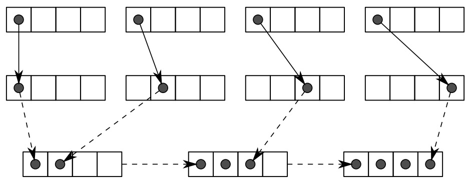
We assume the following informal semantics for instructions. load and store are packed vector load and store instructions. permute and blend are masked permutation and blending instructions. permute moves the ith lane of the source to the lane of the target given in the ith lane of the mask. blend selects the ith lane of the target from the left source if the ith mask lane is L, or from the right source if it is R.

Presentation. We write mask literals as a list of lane values enclosed in angled brackets. Lanes containing the special $*$ mask element indicate a don't-care output in the lane. For example, the instruction permute a, $\langle *, 0, *, 1\rangle$, b moves lanes 0 and 1 of vector a to lanes 1 and 3 of vector b, and leaves lanes 0 and 2 in an undefined state. In addition to the $*$ element, masks for the blend instruction may contain only the two special values L and R, indicating left and right source register, respectively. In graphical figures where data movement is indicated with arrows, an arrow with a solid line represents data movement using the `permute` instruction, and an arrow with a dashed line represents data movement using the `blend` instruction.

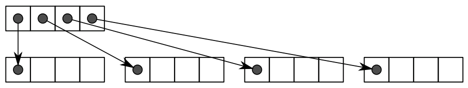
2.2. Enabling Interleaved Access: Automatically Vectorizing a Single Strided Access

We vectorize a nonconsecutive read by first mapping the memory range accessed end-to-end into vector registers (the load-mapped register set). By permuting and blending this register set together, we can extract any subset of up to `VF` lanes into a single packed vector register. We vectorize nonconsecutive writes similarly, by first mapping a store-mapped register set to the memory region being written. A non-consecutive write of data in a packed vector register is performed by expanding the register into a register set with items at the correct locations, and then combining this register set into the store-mapped register set using the `blend` instruction. It is important to stress that these registers are only logically mapped to memory — a mapped register will result in the generation of a memory operation only if one or more lanes in the register are active.

Any vectorized strided access may touch elements in the range of $(\texttt{stride} * \texttt{VF})$ consecutive memory locations in one vectorized loop iteration. Since this memory region is mapped by registers of length `VF` elements, it follows that a maximum of `stride` mapped registers are required for a single vectorized access. Figures 2a and 2b show graphically the action of this simple canonical technique, and Algorithms 1 and 2 contain the logic required to generate the depicted instruction sequences.



(a) Algorithm 1. Nonconsecutive read of the form $\texttt{a}[4 * \texttt{i}]$ with $\texttt{VF} = 4$. First we `permute` the load-mapped register set so that selected lanes do not collide under vertical composition with `blend`. We then `blend` the permuted registers together to form the packed vector corresponding to the access.



(b) Algorithm 2. Nonconsecutive write of the form $\texttt{a}[4 * \texttt{i}]$ with $\texttt{VF} = 4$. We expand the register to be stored into a register set which shadows the array region to be written. This store-mapped register set is then written over the shadowed array region with predicated writes, or using a read-modify-write sequence.

Fig. 2: Vectorizing interleaved access using mapped register sets.

Our approach has two phases: one phase permutes the mapped register set to eliminate lane collision when interleaving or deinterleaving, and the other phase takes a collision-free register set and combines registers using the `blend` instruction to form a packed result. To see why lane collision is a problem, consider Figure 2a: we cannot directly `blend` the initial mapped registers together, because multiple elements occupy the same lane in their respective registers, and collide when using the `blend` instruction.

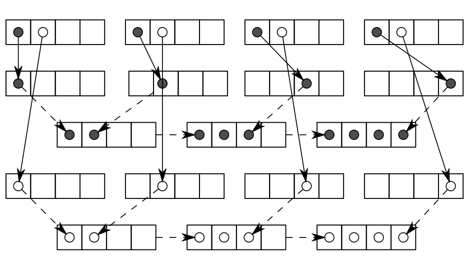
2.3. Exploiting Spatial Locality: Grouping Multiple Interleaved Accesses

Multiple accesses to the same source or destination array can require overlapping vector loads or stores in a vectorized loop iteration if they share the same stride of access (shared-stride accesses). When this is the case, the accesses often exhibit spatial locality which can be exploited to reduce the number of memory operations in the vectorized program. Using the simple canonical approach from Section 2.2, we might generate loads and stores of the same data more than once. Similar to Nuzman et al. [2006], we exploit this spatial locality by allowing multiple accesses to share mapped register sets when interleaving/deinterleaving, reducing the number of memory operations in the vectorized loop.

The degree of spatial locality present between any two shared-stride read or write accesses to the same array in vectorized loop iteration ($i/\texttt{VF}$) depends on the distance between accessed array elements in scalar loop iteration i, that is, the absolute difference between the `offset` of the two access functions. For any two shared-stride accesses with distinct offsets, three scenarios are possible.

(1) *No locality* — When the distance between offsets is greater than or equal to $(\texttt{stride} * \texttt{VF})$, the accesses do not overlap vector loads or stores in a vectorized loop iteration.
(2) *Partial locality* — When the distance is strictly less than $(\texttt{stride} * \texttt{VF})$ and greater than or equal to `stride`, there is partial reuse — some elements of the first access will map to the same loaded or stored registers as elements of the second.
(3) *Full locality* — When the distance is strictly less than `stride`, there is full reuse in a vectorized loop iteration — all `VF` elements of each access map to the same set of vector loads or stores.

Note that our definition does not take into account temporal locality found along the backedge of the vectorized loop. Rather, we focus exclusively on exploiting spatial locality within `VF` iterations of the original loop.


(a) Algorithm 1. Two nonconsecutive reads of the form $\texttt{a}[4 * \texttt{i}]$ (dark dots) and $\texttt{a}[4 * \texttt{i} + 1]$ (light dots) with $\texttt{VF} = 4$, with shared spatial locality can be serviced from the same load-mapped register set using `permute` and `blend` sequences.

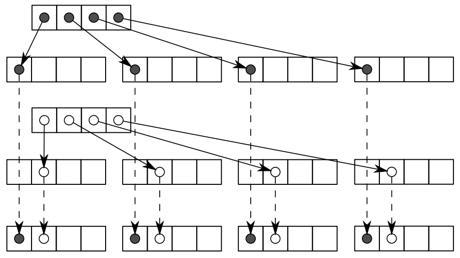

(b) Algorithm 2. Two nonconsecutive writes of the form $\texttt{a}[4 * \texttt{i}]$ (dark dots) and $\texttt{a}[4 * \texttt{i} + 1]$ (light dots) with $\texttt{VF} = 4$ are composed into a single store-mapped register set, reducing the number of stores required.

Fig. 3: Using shared load-mapped/store-mapped register sets to exploit spatial locality.

Consider the pair of shared-stride memory accesses $\texttt{a}[4 * \texttt{i}]$ and $\texttt{a}[4 * \texttt{i} + 1]$, from Figure 3a or 3b. This pair of accesses require the same vector memory operations in every vectorized loop iteration — the grouping exhibits *full locality* as we have defined it. However, when we include an access $\texttt{a}[4 * \texttt{i} + 5]$, it will require an extra memory operation in every vectorized loop iteration, because the underlying memory regions do not completely overlap (*partial locality*). Including another access, $\texttt{a}[4 * \texttt{i} + 7]$, that access would exhibit full locality with access $\texttt{a}[4 * \texttt{i} + 5]$, but only partial locality with accesses $\texttt{a}[4 * \texttt{i}]$ and $\texttt{a}[4 * \texttt{i} + 1]$.

In order to minimize the number of vector memory operations for any number of shared-stride accesses, it is sufficient to consider only groups of maximal size, and load or store each resulting mapped register exactly once. However, minimizing the number of memory operations does not guarantee the generation of optimal vectorized code for a loop containing such accesses, which is a difficult optimization problem.

Considerations include not only repeated memory operations, but also vector register spills and reloads due to register pressure and the scheduling of the instructions which perform the interleaving/deinterleaving, which can exhibit significant instruction-level parallelism with the computation present in the loop. Barik et al. [2010] demonstrate that good solutions to such problems require tight integration between vectorization, register allocation, and instruction scheduling during compilation.

We do not attempt to solve this optimization problem, but use the same simple, practical grouping heuristic for accesses with locality as Nuzman et al. [2006] — group only those accesses which exhibit *full locality* as we have defined it. This approach yields significant speedup in practice, and does not require much engineering effort on the part of the compiler implementor.

In order to decide at compile time which shared-stride accesses can be serviced from a shared mapped register set, we introduce the analytic concept of an access group, which generalizes the similar concept of Nuzman et al. [2006].

DEFINITION 2 (ACCESS GROUP).
Accesses to the same array with the same direction (read or write) and a shared stride, may be grouped by mapping each access offset to some interval $[k, k + (\texttt{stride} - 1)]$*, for* $k \in \mathbb{N}$*, where* $k \equiv 0 \mod \texttt{stride}$*. Each such interval, for any particular stride, defines a distinct access group at that stride. The size of an access group (written* n*) is bounded above by the shared stride of access of the group.*

Let us assume we are considering two accesses $a_0(i) = b_0 + u_0 * (\texttt{stride}_0 * i + \texttt{offset}_0)$ and $a_1(i) = b_1 + u_1 * (\texttt{stride}_1 * i + \texttt{offset}_1)$. These accesses can share the same load or store mapped register set iff

$$\begin{gathered}\texttt{stride}_0 = \texttt{stride}_1 \\ u_0 = u_1 \\ \lfloor (b_0 + u_0 * \texttt{offset}_0)/u_0 * \texttt{stride}_0 \rfloor = \lfloor (b_1 + u_1 * \texttt{offset}_1)/u_1 * \texttt{stride}_1 \rfloor\end{gathered}$$

This formulation groups accesses where stride of access and unit size are equal, and the accesses are relatively aligned within `stride` elements of the shared unit size of access. These criteria are sufficient to ensure that grouped accesses exhibit *full locality*. To vectorize such an access group, we repeatedly apply Algorithm 1 or 2 but use only a single, shared mapped register set. Composing reads and writes into a shared mapped register set reduces the number of memory operations required for any group of n accesses. Since n is bounded above by the stride of access, and the number of mapped registers is exactly equal to the stride of access (Section 3.2), this sharing represents a reduction from worst-case $O(n^2)$ memory operations considering individual accesses to $O(n)$ operations considering the access group.

2.3.1. Dealing with Store-Side Gaps. The work of Nuzman et al. [2006] specifically excludes interleaved access patterns with store-side gaps. A *gap* is any unread or unwritten area of memory between elements of an interleaved access. Figure 3 displays two scenarios with gaps. In both examples in the figure, only two of the four lanes in each loaded or stored vector are used. While unused loaded lanes can simply be discarded, unused lanes in stores require the implementation to preserve the contents of memory in those lanes. As indicated in Figure 2, this may be achieved using predicated writes, or using a read-modify-write sequence. On both of our experimental platforms, predicated writes have very poor performance, while read-modify-write has excellent performance.

While predicated writes typically have the same semantics as the original scalar writes, any implementation using read-modify-write sequences may encounter race conditions due to the fact that a read-modify-write sequence modifies memory elements which were not modified by the scalar code. If the contents of memory corresponding to unused lanes changes between the read and write step of a sequence, data races and memory corruption can result. Broadly, there are three scenarios for such races: races between accesses in the same access group, races between different access groups, and thread-level races between multiple instances of the vectorized code operating on the same memory region.

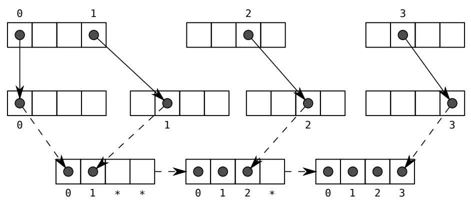
Our approach avoids races between accesses in the same access group by composing all the resulting writes into a single store-mapped register set. Since there is only one read-modify-write sequence, and all writes are required to be non-overlapping, races between the accesses in any one group are avoided by this approach. To avoid inter-access group races, implementations which cannot use predicated writes may place read-modify-write sequences resulting from different access groups into atomic sections or use memory fences to ensure exclusive access to contested memory regions. The same approach may be used to avoid thread-level races. Where the hardware does not provide a way to ensure atomic execution of a group of instructions, or to create memory barriers, read-modify-write cannot always be used safely in a multithreaded context.

3. OPTIMIZATION

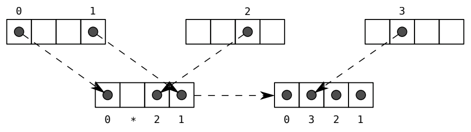
There is significant scope for optimization of the instruction sequences generated by the approach outlined in Section 2. In this section, we present four optimizations which transform the `permute`/`blend` sequence programs generated by our technique.

These optimizations can be broken down into two categories: either reducing the number of permutations in a program, or reducing the number of blends. The optimizations concerning permutation follow from the realization that we typically permute a register for one of two reasons: either we need to eliminate a lane collision for blending, or we need to enforce matching element order in two vector registers to ensure a legal vectorization. In both of these scenarios, we can eliminate permute instructions under some conditions. Concerning blends, we state a property of blend instructions assuming the informal semantics in Section 2 which allows us to merge multiple blend instructions into a single blend instruction. We also introduce a reassociation of blend instruction sequences to increase the count of mergeable instructions under this property.

3.1. Eliminating Permutations I: Executing Original Loop Iterations Out-Of-Order



(a) Algorithm 1. Memory access of the form $\texttt{a}[3 * \texttt{i}]$ with $\texttt{VF} = 4$. $*$ denotes a don't-care element in an output register. Elements are labelled with their index in original program order.



(b) Algorithm 3. Performing the deinterleaving step without first permuting the load-mapped register set to eliminate lane collision under vertical composition with the `blend` instruction. Memory access of the form $\texttt{a}[3 * \texttt{i}]$ with $\texttt{VF} = 4$. Elements are labelled with their index in original program order.

Fig. 4: Effect of reordering original loop iterations to match data layout.

The lane-collision removing permutation stage of Algorithms 1 and 2 is unneccesary when the data layout in a mapped register set is already free of lane-collisions. When the data layout is collision-free, we can obtain a large savings on data reorganization by skipping the permutation stage and directly blending mapped registers together (Figure 4). However, this approach can cause packed results to be produced with elements out of order with respect to the original loop (Figure 4b). If each iteration of the original loop was independent, then it is permissible to reorder operations within a SIMD instruction. To obtain a legal vectorization of the original program, care must be taken to maintain matching orders of operations within the SIMD instructions in the vectorized loop. To ensure a legal vectorization, we must introduce a permutation whenever the order of elements in different operands of the same SIMD instruction do not match. We refer the reader to Figure 5a for a detailed example.

Determining the order of execution of scalar loop iterations within a vectorized loop iteration to minimize the overall number of permutations is an optimization problem which appears hard. The number of possible iteration orderings is the factorial of the vectorization factor, and each ordering implies a (possibly identity) permutation of every register which is the target of a gather or the source of a scatter. We do not attempt to solve the problem in this article. Instead, to keep the number of permutations reintroduced small, we apply a simple, practical heuristic. We examine the data flow graph of the vectorized program, and choose the most commonly observed element order of deinterleaved data elements as the order in which to execute the loop iterations. We verify that this heuristic is sufficient to achieve significant speedup in practice (Section 4.4). Having chosen this shared order, we apply Algorithms 3 and 4 to generate vectorized interleaving or deinterleaving code for each access, then scan the vectorized program, reintroducing permutations where necessary to enforce the chosen order of operation and ensure a legal vectorization.

3.2. Eliminating Permutations II: Simultaneously Resolving Collisions for Multiple Accesses

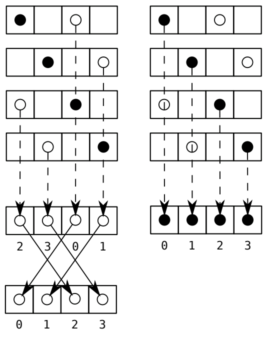
When the data layout in a mapped register set is not free of collisions, we cannot apply the optimization detailed in Section 3.1 to skip the lane-collision removing permutation stage of Algorithms 1 and 2. However, if we must perform some permutations to remove lane collisions, we can avoid permuting the entire register set for each access, as in Algorithms 1 and 2.

The goal is to choose, for each colliding access, a unique lane number in the range of `VF` lanes for each of the `VF` elements of the access. Let us say that two registers collide if *any* access occupies the same lane in both registers. In order to remove *all* collisions between the two registers, we can logically rotate one register by increments until our unique lane numbering condition is achieved for every contained access. If we extend this transformation to the full register set, so that element-wise vertical collision between each register and all registers with a lower index is removed, we have eliminated lane collision for all accesses in the contained access group with only one rotation of each register. As long as the group contains fewer than or exactly `stride` accesses, each vector lane holds at most one accessed element, which ensures that the transformation is possible. This property is ensured by our definition of an access group from Section 2 which sets the upper bound on the number of grouped accesses to exactly `stride`. Logical rotation is achieved using the `permute` instruction with a mask which arranges elements in rotated order. Since the transformation ensures that the mapped register set is free of lane collisions, it is always possible to apply Algorithms 3 and 4 immediately afterwards. Algorithms 5 and 6 state this combined approach.

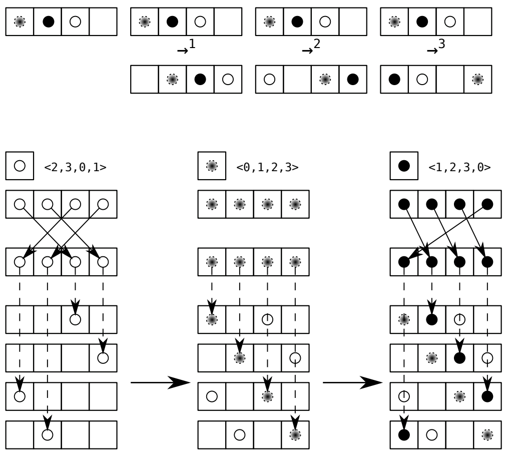
Figure 5a shows graphically the action of Algorithms 5 and 6. If the accesses are reads, then we can simply rotate each load-mapped register immediately after it has been loaded. However, for writes, the transformation is a little more subtle. For writes, rather than simply transforming a register set from one order to another, we are creating a register set in *transformed* order by combining registers with the `blend` instruction. To interleave a register resulting from computation into the store-mapped register set, we blend it with each store-mapped register in turn, with each blend forming a new store-mapped register by inserting one or more elements of the compute register. Since we are composing registers with the `blend` instruction, we must ensure that the access does not have lane collisions, which would require multiple stored elements to map to single lanes of the compute register.

Our approach runs as follows. We first compute what rotation of each store-mapped register is required so that each element of every access occupies a unique lane in the store-mapped register set. This gives us the *rotated store order* we must produce. Next, for each access to be interleaved, we take the lane number in our rotated store order of each element of that access. This gives us the required order of elements in the source register so that it can be blended into the store-mapped register set without collision.

Finally, each register resulting from computation is permuted into the required order before applying the `blend` sequence which combines it with each store-mapped register. Figure 5b shows this graphically. Intuitively, blending together these permuted registers results in the generation of a rotated image of the store-mapped register set. We then perform an inverse rotation of each store-mapped register before storing to achieve the target store order. This scheme guarantees the introduction of at most one permutation per mapped register, plus at most one permutation per compute register to be stored, as opposed to the naive approach presented in Algorithms 1 and 2, which permutes every mapped register at least once for every access which shares it.



(a) Algorithm 5. Our transformation enables vertical composition with the `blend` instruction of colliding memory accesses with a single transformation of the mapped register set. One access is out-of-order under vertical composition with `blend` after data layout transformation (left side). However, the vectorization can be legalized with a single `permute` as shown, to ensure the order of elements in each register is the same.



(b) Algorithm 6. Accesses $\mathtt{a}[4*\mathtt{i}+2]$ (light dots), $\mathtt{a}[4*\mathtt{i}]$ (patterned dots), and $\mathtt{a}[4*\mathtt{i}+1]$ (black dots). Desired store order at top with rotated store order underneath. Required permutation of each access to obtain rotated store order indicated with permute masks. Heavy arrows at bottom show the evolution of the store-mapped register set as each access is interleaved in. Final store-mapped register set before inverse rotation is shown at bottom right.

Fig. 5: Simultaneously resolving lane collisions for an entire vectorized access group. The access group shown in 5a is a read group, with a write access group shown in 5b.

Statically Determining Lane Collisions. The presence or absence of lane collisions can be statically determined. To see why this is the case, consider any scalar access $a(\mathtt{i})$: the memory region touched by the vectorized access is divided by our approach into buckets of VF consecutive elements. The extent of the memory region is $(\mathtt{stride} * \mathtt{VF})$ scalar elements, and the index within the region of element i of the scalar access is given by $(\mathtt{stride} * i + \mathtt{offset})$. The corresponding vector lane number of the element is found by floor division of this index by VF. It follows that the elements of any access will repeat vector lane numbers, and collide when composed with the blend instruction, after $lcm(\mathtt{VF}, \mathtt{stride})$ scalar elements. Lane collision occurs when the length of the memory region touched by the vectorized access is larger than this quantity, i.e. when $(\mathtt{stride} * \mathtt{VF}) > lcm(\mathtt{VF}, \mathtt{stride})$. By definition, the condition is false when stride and VF are coprime.

3.3. Reassociation of Blend Instructions

Formally, our blend instruction can be represented as a binary operator $\oplus_m$, denoting the result of blending left and right operands with the mask m. Under this definition, the expression $(((a \oplus_m b) \oplus_{m'} c) \oplus_{m''} d)$ with initial masks m, m', m'', is equivalent to the reassociated expression $((a \oplus_n b) \oplus_{n'} (c \oplus_{n''} d))$ for some reassociated masks n, n', n''. Reassociation causes blend masks to change because the operands of the individual blend instructions are exchanged. Figure 6 states the formal rewrite rule for blend instructions in the vectorized code, with computation of reassociated blend masks n, n', n''.

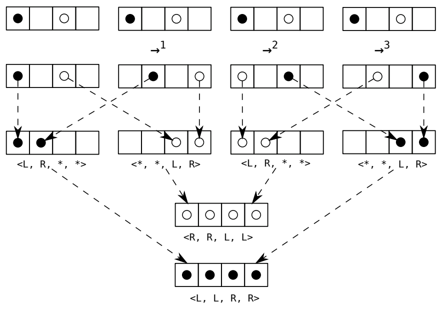
$$\begin{array}{c}
(((a \oplus_m b) \oplus_{m'} c) \oplus_{m''} d) \\
\hline
n \leftarrow m, \qquad n'' \leftarrow \{(i,\ L) \mid (i,\ x) \in \mathit{rights}(m')\} \cup \{(i,\ R) \mid (i,\ x) \in \mathit{rights}(m'')\} \\
\hline
n' \leftarrow \{(i,\ L) \mid (i,\ x) \in \mathit{active}(n)\} \cup \{(i,\ R) \mid (i,\ x) \in \mathit{active}(n'')\} \\
\hline
((a \oplus_n b) \oplus_{n'} (c \oplus_{n''} d))
\end{array}$$

Fig. 6: Rewrite rule for reassociation of blend instruction sequences. The logical operation $\mathit{rights}(m)$ for some mask m selects all mask lanes which contain the R selector. The logical operation $\mathit{active}(m)$ selects all mask lanes which contain either L or R selectors.

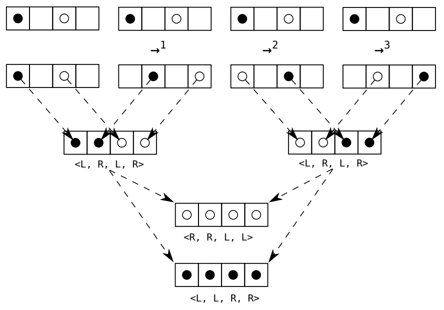
Reassociation of a blend reduction sequence transforms dependence structure. The sequences initially produced by our approach use a single register as an accumulator, blending in lanes from one register at a time to form a packed result. This approach results in low register pressure, requiring only a single live register to accomodate intermediate results, but requires sequential execution even when blend instructions can be independent. Fully reassociated blend sequences contain the same number of instructions, but perform the work as a parallel binary reduction. This approach has a high degree of instruction level parallelism, but increased register pressure versus the sequential reduction.

3.4. Eliminating Blends: Merging Multiple Blend Instructions

This section presents a novel optimization for blend reduction sequences of the sort generated by our technique. The key observation is that one vector register may hold the result of two different blend instructions if certain conditions are met. This allows us to merge multiple blend instructions into a single blend instruction, resulting in faster generated code with reduced register pressure. Intuitively, two blend instructions with identical left and right sources may be merged into a single blend instruction if the set of active output lanes of the two instructions are disjoint. The resultant register simultaneously carries the definition of both results of the initial pair of blend instructions. The resultant merged blend instruction may itself be merged with other blend instructions, and this merging may continue until all output lanes of the instruc-



(a) Algorithm 5 plus `blend` reassociation before merging blends: 6 `blend` instructions, 4 temporary registers



(b) Algorithm 5 plus `blend` reassociation, after merging blends: 4 `blend` instructions, 2 temporary registers

Fig. 7: Graphical depiction of data flow before and after merging `blend` instructions. Two reads of the form a$[4 * \mathtt{i}]$ (dark dots) and a$[4 * \mathtt{i} + 2]$ (light dots). The load-mapped register set (top line of diagrams) is rotated as indicated to remove lane collisions (second line of diagrams). Masks for `blend` instructions are indicated below each result.

tion are active. The merging can be defined as the result of a simple algebraic rewrite rule which may be applied repeatedly to the program to merge blend instructions.

When stating program transformations as rewrite rules, the top line represents the initial program fragment, which is the pattern that must be matched to trigger the rule. The remaining lines represent necessary conditions which must hold for the rule to be applied. The final line represents the modified program fragment after the application of the rewrite rule. We represent masks in rewrite rules as indexed sets of mask elements. We say that two blend masks are disjoint iff everywhere there is an active lane in one, there is a corresponding don't-care lane in the other. Figure 7 shows graphically the action of this `blend`-merging rewrite rule.

$$\begin{array}{c}
\texttt{blend r0, r1, mask1, r2} \qquad \texttt{blend r0, r1, mask2, r3} \\
\hline
\{i \mid (i,\ x) \in mask1 \wedge x \neq *\} \cap \{j \mid (j,\ x) \in mask2 \wedge x \neq *\} = \emptyset \\
\hline
mask3 \leftarrow (\{(i,\ mask2[i]) \mid (i,\ *) \in mask1\} \cup \{(i,\ mask1[i]) \mid (i,\ *) \in mask2\}) \\
\hline
\texttt{blend r0, r1, mask3, rNew}
\end{array}$$

Fig. 8: Rewrite rule for merging `blend` instruction pairs.

4. EVALUATION

4.1. Time Complexity of Generated Code

In Sections 2 and 3 we have presented three approaches for vectorization of interleaved memory reads and writes with arbitrary constant strides. Each approach results in the generation of a fixed number of `permute` and `blend` instructions. In order to facilitate compile-time decision making about which approach to use, we present an analysis of the time complexity of generated code in terms of the number of instructions generated by each of these techniques.

Simple Canonical Technique (Algorithms 1 and 2)

As described in Section 2.2, the size of the mapped register set for any access is at most `stride` vector registers. When `stride` $>=$ `VF`, at most `VF` registers are required to be combined to vectorize any one access. In this case, each active mapped register contains only a single data lane required by the vectorized access. Proceeding according to Algorithms 1 and 2, each vectorized access requires `VF` permutations (one per active mapped register). After reassociation, each packed register resulting from interleaving or deinterleaving is at the root of a full binary tree of blend operations with VF leaves. However, reassociation does not change the number of blend operations, which remains at $\texttt{VF} - 1$ blends per access. The number of generated instructions for a single interleaved access is therefore $2\texttt{VF} - 1$ operations using this approach.

Out-of-Order Technique (Algorithms 3 and 4)

When a strided access satisfies the alignment criterion of being collision free at VF (Section 3.1) we may apply Algorithms 3 and 4 to vectorize it. We exploit the independence of original loop iterations to change the scalar iteration order within a single vectorized loop iteration. This tactic allows us to choose an iteration order which reduces the overhead of data layout transformation. Following Algorithms 3 and 4, for any one strided access we generate the tree of $\texttt{VF} - 1$ blend operations to combine the elements of the access into a single register. We then inspect each packed register and determine the most common iteration order implied by the results. In the worst case, we reintroduce a permutation for every packed result to legalize the vectorization. The number of generated instructions is VF for each vectorized access.

Collision Resolving Technique (Algorithms 5 and 6)

Although the asymptotic complexities of the simple canonical approach and the out-of-order approach are both of order $O(\texttt{VF})$ for a single access, the number of generated permutes and blends for the out-of-order technique is approximately half that of the canonical technique (VF versus $2\texttt{VF} - 1$). To reduce the number of generated instructions, the compiler should attempt to apply the out-of-order technique if it is applicable. Algorithms 5 and 6 introduce a method for applying the out-of-order technique for any access conforming to Definition 1. As detailed in Section 3.2, the cost of the transformation is amortized when an access group contains more than one access. The total count of operations using Algorithms 5 and 6 for a group of n shared-stride accesses is `stride` permute operations to resolve lane collisions followed by VF operations per access to perform vectorization, for a total of $(n * \texttt{VF}) + \texttt{stride}$ operations to vectorize n shared-stride accesses. We summarize our analysis in Table I.

Technique	Instructions Generated	Order
Algorithms 1 and 2	$n * (2\texttt{VF} - 1)$	$O(n * \texttt{VF})$
Algorithms 3 and 4	$n * \texttt{VF}$	$O(n * \texttt{VF})$
Algorithms 5 and 6	$(n * \texttt{VF}) + \texttt{stride}$	$O(n * \texttt{VF})$

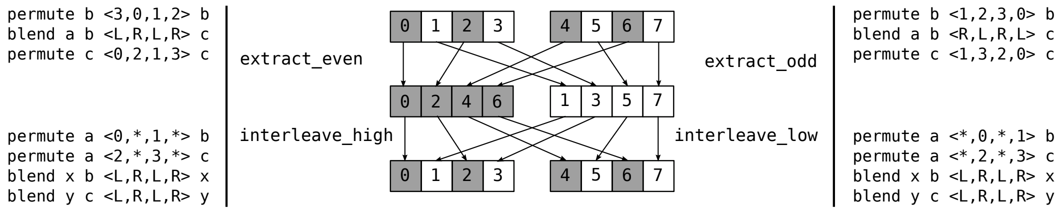
Table I: Time complexity (number of generated instructions) of SIMD interleaving and deinterleaving code generated by the proposed techniques, for a group of n accesses at a shared stride of `stride` elements.

Table I omits the effect of our blend merging transformation from Section 3.4. Arithmetic properties of the stride and offset of each access, and VF determine the contents of masks in the tree of `blend` instructions generated by our approach. Because of this, the effect of blend merging is highly dependent on the input program. However, accesses vectorized using our approach will often result in trees of blend instructions with a high degree of compatibility. These trees exhibit the property that `blend` instructions at corresponding locations in two trees are pairwise mergeable. In such cases, the original pair of trees can be merged up to the root instructions, which cannot be merged because they each produce a full output register after merging their subtrees. Figure 7 shows one of these programs. The original pair of trees have a combined count of $(2\texttt{VF} - 2)$ instructions before merging, and the merged tree contains VF instructions. Real-world benchmark program cxdotp − 2D (Figure 12) exhibits this property, and blend merging has a pronounced effect.

It is not possible to state the effect of blend merging on time complexity for a single access, because blend merging amortizes the total cost of blending over a group of accesses. We can characterize the effect of the transformation on an access group with an extra assumption. When an access group is full, by definition, accesses with all n distinct offsets are present. When n is even, each tree of blend instructions can be merged with exactly one other tree, and $n/2$ blend sequences result from blend merging. When n is odd, one instruction tree cannot be paired, and $n/2+1$ sequences result. The number of blend instructions required to interleave or deinterleave the entire group of n accesses, for $n > 1$, is thus $(n/2) * \texttt{VF}$ for even n, and $(n/2+1) * \texttt{VF}$ for odd n. Blend merging does not change the asymptotic complexity, which remains of order $O(n * \texttt{VF})$.

4.2. Comparison with Nuzman et al.

The technique of Nuzman et al. [2006] generates extremely efficient code for interleaved access with power-of-two strides. However, the approach can only handle powers of two — when the stride is not a power of two, the technique of Nuzman et al. is not applicable. We generalize the approach of Nuzman et al. to arbitrary constant strides. Nuzman et al. use an intermediate representation with a small number of primitives, shown in Figure 9. These primitives precisely express interleaving/deinterleaving where the stride is a power of two. Our representation uses more generic primitives, which can express interleaving/deinterleaving at any constant stride, but require a constant factor more operations for power-of-two stride. This constant factor is demonstrated by the direct correspondence between each of the primitives of Nuzman et al. and a short, fixed sequence in our representation. The sequence corresponding to each Nuzman primitive is indicated in Figure 9, for $\texttt{VF} = 4$. This correspondence between representations often leads to identical native code after instruction selection when the stride is a power of two, because the native instructions implementing the primitives of Nuzman et al. are also selectable for the corresponding sequence in our representation.



```
permute b <3,0,1,2> b
blend a b <L,R,L,R> c
permute c <0,2,1,3> c
```

```
permute b <1,2,3,0> b
blend a b <R,L,R,L> c
permute c <1,3,2,0> c
```

```
permute a <0,*,1,*> b
permute a <2,*,3,*> c
blend x b <L,R,L,R> x
blend y c <L,R,L,R> y
```

```
permute a <*,0,*,1> b
permute a <*,2,*,3> c
blend x b <L,R,L,R> x
blend y c <L,R,L,R> y
```

Fig. 9: Primitive operations of Nuzman et al. extract_even and extract_odd extract in-order even or odd indexed lanes from two source registers. interleave_high and interleave_low perform the inverse data movements. Corresponding operations in our representation shown to the left and right of the diagram, for $\texttt{VF} = 4$.

The number of generated instructions in vectorized code from the technique of Nuzman et al. for power-of-two strided accesses is of the same order as our proposed techniques. As detailed in Nuzman et al. [2006], their technique generates a perfect, complete binary tree of instructions of $log_2(\delta)$ levels for a vectorized interleaved memory access, where δ is the stride of access. The trees generated are perfect and complete because the technique only considers programs where δ is a power of two, and each level is formed by combining pairs of adjacent inputs from previous levels [Rosen 2011]. The $\delta - 1$ generated instructions correspond to the reassociated tree of blend operations produced by our approach. However, for any one vectorized access, the number of vector registers which must be combined is bounded above by VF, by the same argument as for our approach (Section 4.1). The maximal instruction count is thus $\texttt{VF} - 1$, making the asymptotic complexity of extraction of a group of n accesses order $O(n * \texttt{VF})$ using either technique.

4.3. Native Code Generation

In this article we address the problem of vectorizing interleaved access, and propose an approach which can generate vectorized code for accesses with arbitrary constant strides. However, generating correct vectorized programs is only of use if vectorization improves the overall performance of the program. An important step in realizing a performance gain in practice is the generation of fast native code.

In our implementation, we use a modified version of the "Bottom-Up Tree Pattern Matching" approach [Aho et al. 1989; Balachandran et al. 1990], with a simplified cost heuristic. For a subset of the available native data reorganization instructions on our two experimental platforms, we derived a table mapping each native instruction to the corresponding tree of abstract operations in our IR. Driven by this table, we perform the tree rewrite by greedily selecting the native instruction which covers the largest available subtree of our abstract operations at each step. It is possible to improve on this approach to instruction selection [Fraser et al. 1992], particularly for vector instructions [Barik et al. 2010], but we found that even this simple approach was sufficient to realize a practical speedup from our techniques in experimental evaluation. Native instruction selection is a large topic, and is not the focus of this article, but future work could involve the use of an optimal instruction selection scheme to increase the performance of code generated by our approach. Possibilities in this direction are discussed in Section 5.3. In particular, GCC's instruction selection for the primitives of Nuzman et al. is very efficient, as can be seen by looking at the powers of two strides in Figures 10a and 10b. However, our simple scheme sometimes makes a better selection even for powers of two (Figure 10b, stride=4).

4.4. Experimental Evaluation

Our benchmarking was carried out on two experimental platforms: we used an Intel Core i5-2500 (Sandy Bridge) system with 16GB of RAM as our 128-bit "SSE" platform, and an Intel Core i5-4570 (Haswell) system with 32GB of RAM as our 256-bit "AVX2" platform. Experiments were run on Linux (kernel version 4.1). We followed the guidelines outlined in Paoloni [2010] for benchmarking short programs on our experimental architecture. Our baseline scalar code was generated by running GCC 5.0 on plain C code, with optimization level *-O3* and vectorization disabled. GCC implements Nuzman's algorithm for vectorization of interleaved access. For comparison with Nuzman, we generated vectorized code using GCC *-O3* with vectorization enabled. We inspected the generated assembly and verified that GCC applied Nuzman's algorithm where the stride is a power of two. We implemented our vectorization techniques in a simple compiler that generates vector intrinsics, and compiled the resulting code with GCC *-O3*.

Figures 10 and 11 present the results of synthetic benchmarking of programs performing data movement on SSE. We generated programs which performed either a gathering operation (Figure 10) or a scattering operation (Figure 11). We present the speedup achieved by our simple, canonical approach using Algorithms 1 and 2 and our reordering approach using Algorithms 3, 4, 5 or 6 as appropriate. In all cases, the stride of access was swept through the range $[2, 16]$ — this choice was influenced by the experimental architecture (SSE), which has 16-byte vector registers. Where the stride of any individual gathering or scattering operation exceeds 16 bytes, it must perform at least as many vector memory operations as there were scalar memory operations in the original loop. On our SSE experimental platform, regardless of the technique employed, we would expect the performance of vectorized memory access to degrade as stride length increases.

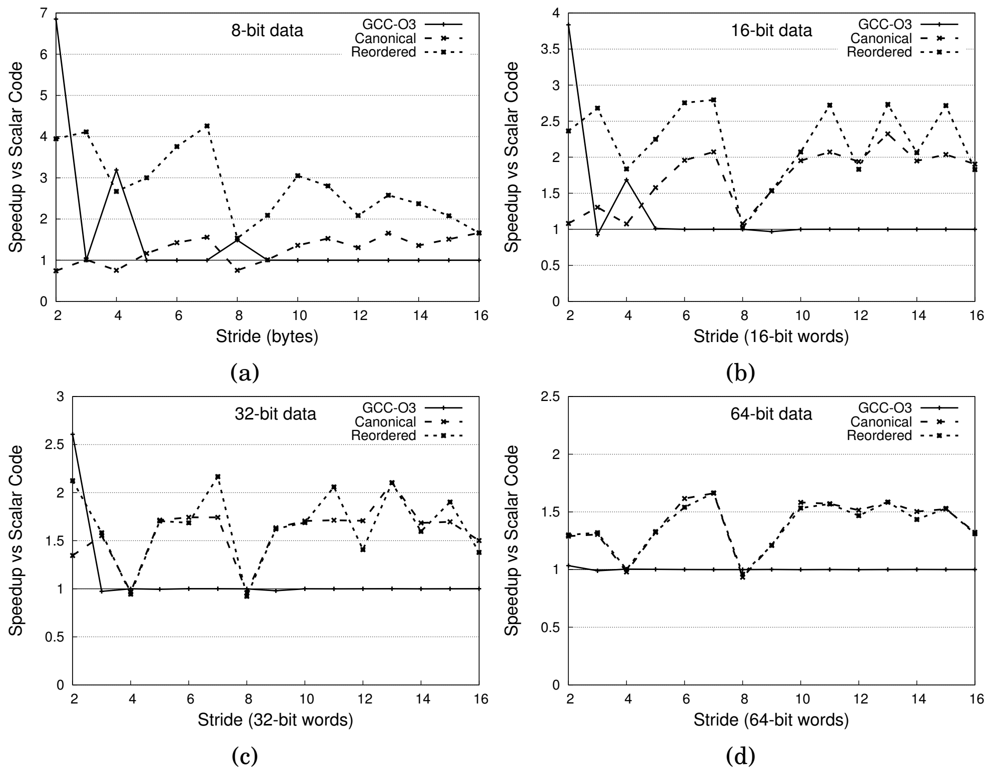


Fig. 10: Speedup over scalar code for a single strided read (gather).

Discussion of Results

Performance Limits. Performance degradation with increasing stride is visible in Figure 11, but only up to the length of an architectural vector register. Note that the stride of access in each graph is given in units of the word size, so a stride of 16 bytes is exceeded very quickly for wider types. In each case, when the stride exceeds 16 bytes, the performance of vectorized memory access is reduced to parity or near-parity with the scalar code, but does not further degrade with increasing stride, up to a stride of 128 bytes (the largest experimental value). Further, the experiments show that in general, large speedups are possible for single strided accesses where the stride is shorter than a vector register. The performance drop at strides longer than a vector register is significantly less pronounced for gathering operations than for scattering operations. For example, our approach achieves 1.66x speedup versus scalar code performing a 64-bit stride 7 gather operation (Figure 10d). The stride of this operation is 56 bytes, much longer than a 16-byte vector register on our SSE experimental platform. The strategy of tiling a memory region with vector loads and composing required elements into results with SIMD instructions appears particularly effective on SSE. For 8-bit stride 3 gather operations (Figure 10a) our approach results in more than 4x speedup over scalar code, but GCC cannot vectorize the program (the approach of Nuzman et al. is not applicable). This case of data movement is ubiquitous in image and video processing applications, where formats using packed 8-bit triples are common.

Effect of Reordering. The most pronounced difference between gathering (Figure 10) and scattering (Figure 11) results is in the effect of reordering loop iterations to re-

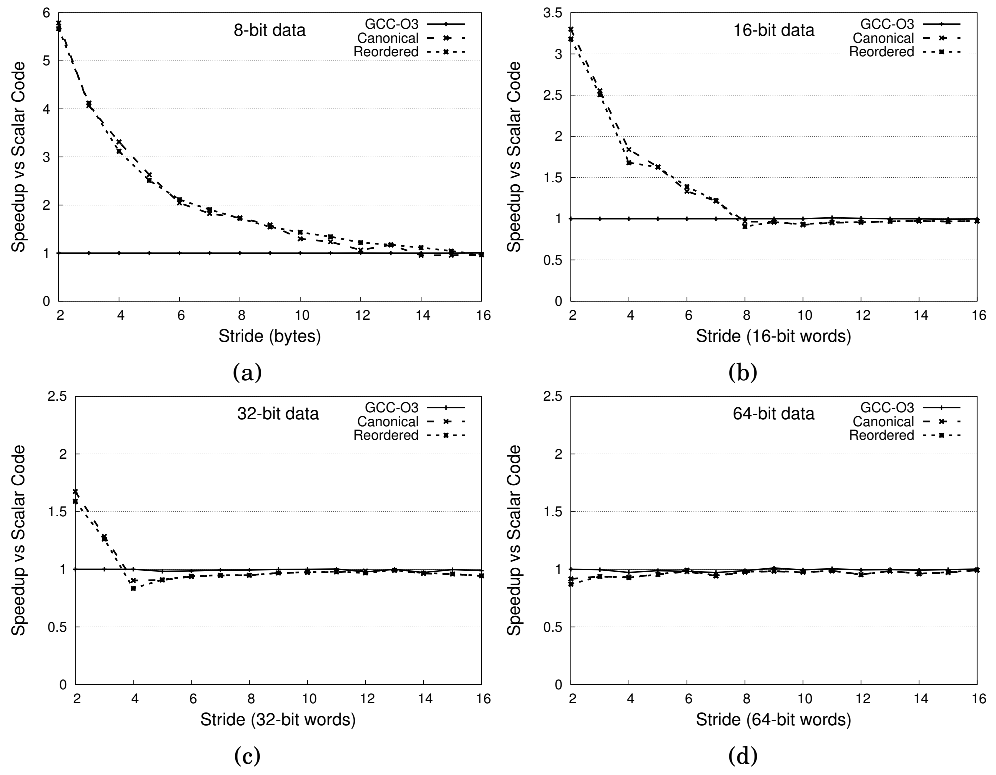


Fig. 11: Speedup over scalar code for a single strided write (scatter).

duce permutation overhead of interleaved access. When scalar data elements are positioned so that there is no lane collision while blending, we can apply Algorithms 3 and 4 and forego the permutation phase which removes lane collisions, resulting in a shorter program. Although analytically the reduction in number of generated instructions is equivalent for both Algorithm 3 and 4, a key semantic difference is that vector loads with unused lanes do not require different treatment from loads without, whereas vector stores with unused lanes must preserve the contents of memory between stored elements. We implemented such stores using a read-modify-write sequence, using our `load`, `blend` and `store` instructions. For stores, the relatively long latency of read-modify-write memory access acts to smooth the speedup obtained from improvements in data reorganization.

Relation of Speedup to Stride. When performing vectorized interleaved memory operations, there is often a pronounced difference between our simple canonical approach and reordered approach at neighbouring strides. At any fixed vectorization factor, accesses at a given stride will either exhibit lane collision or will not, determining whether permutations must be introduced to resolve these lane collisions. The presence of lane collisions can be statically determined — lane collisions are not present when `stride` and `VF` are coprime (Section 3.2). On our SSE experimental platform, the natural vectorization factors for the four machine types with distinct bit-width are powers of two ($\texttt{VF} = 16$ for 8-bit data, $\texttt{VF} = 4$ for 32-bit data, and so on), meaning that lane collisions are generally present at even but not odd strides, once `stride` exceeds `VF`. This pattern is observed in the simple oscillation of speedup across neighbouring strides in our synthetic benchmarks, though smoothed in the case of stores as previously noted.

Variability of Scalar Code. We would expect to see the oscillation previously mentioned throughout synthetic benchmarking, but it is often obscured by variability in the scalar code. The performance of the scalar code produced by GCC at any two neighbouring strides can be significantly different. In particular, GCC does very well when optimizing gathering operations for locality, and incorporates several patterns which produce fast code for common or idiomatic memory access patterns. In our synthetic benchmarking, GCC sometimes produces code which is faster than the best vectorized code produced from our approach. We investigated the performance difference and found that GCC chose to vectorize the data movement using Nuzman's algorithm. However, post-pass instruction selection emits scalar code for the abstract operations of Nuzman et al. Performing this devectorization step requires a very detailed cost model, and we did not attempt to replicate it in our experimental compiler. Apart from these cases, the performance of the best vectorized code produced by our approach matched or exceeded the performance of code generated by GCC at optimization level *-O3*. In Figure 10, for some power-of-two strides, the speedups achieved by GCC and our approach are identical. In these cases, both our technique and the technique of Nuzman et al. result in identical native code after instruction selection.

Real-World Benchmarking (SSE)

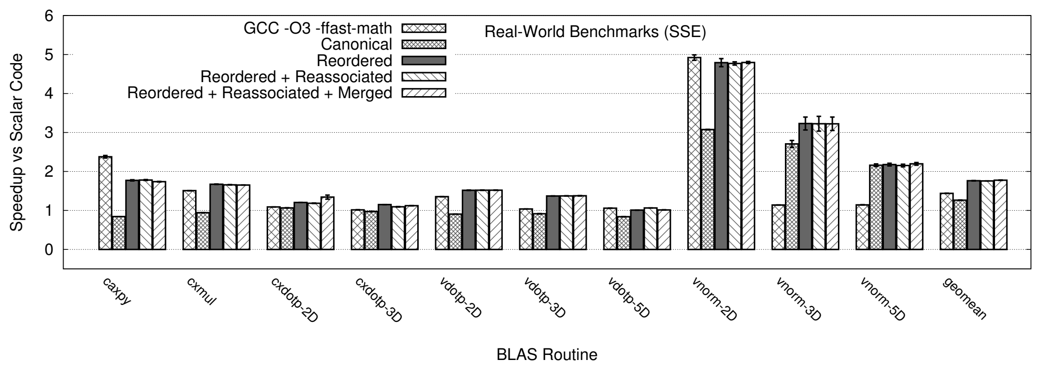


Fig. 12: **SSE**: BLAS Level 1 operations exhibiting interleaved memory access optimized by GCC and using our approaches. Mean speedup over 10,000 runs of each program is plotted, with error bars showing a variance of one standard error about the mean. *geomean* is the geometric mean speedup of each approach over all benchmarks.

Our real-world benchmarking uses a selection of BLAS Level 1 [Lawson et al. 1979] routines with varying access patterns. Figure 12 presents the results on our SSE experimental platform. We display speedup over scalar C code obtained by GCC using optimization level *-O3*, and also using each of the techniques we propose. Details of each benchmark program are listed in Table II. We vectorize computation by simple scalar expansion.

Of the 10 programs, 5 can be vectorized using the technique of Nuzman et al., and GCC applies it in 4 of 5 cases. In the case of caxpy, GCC uses a combination of classical optimizations to transform the program so that memory access becomes consecutive, and vectorizes using scalar expansion. This results in extremely compact code, which achieves a speedup of over 2x versus scalar code. Applying the techniques we have described results in a speedup of 1.8x using reassociation and reordering.

BLAS L1 Routine			Benchmark Instantiation						
			Loads		Stores			GCC Applies	
Type	Name	Dim.	Stride	#	Stride	#	C Type	SIMD	Nuzman
cx	axpy	1D	2	4	2	2	`float`	Yes	No
cx	mul	1D	2	4	2	2	`float`	Yes	Yes
cx	dotp	2D	4	8	2	2	`float`	Yes	Yes
cx	dotp	3D	6	12	2	2	`float`	No	N/A
s	dotp	2D	2	4	1	1	`float`	Yes	Yes
s	dotp	3D	3	6	1	1	`float`	No	N/A
s	dotp	5D	5	10	1	1	`float`	No	N/A
s	norm	2D	2	2	1	1	`float`	Yes	Yes
s	norm	3D	3	3	1	1	`float`	No	N/A
s	norm	5D	5	5	1	1	`float`	No	N/A

Table II: Summary of dimensions, strides, underlying C types, and vectorization realized for each instantiation of the BLAS Level 1 routines used in benchmarking in Figure 12. For example, cxdotp-3D is the dot-product of 3-dimensional vectors of complex numbers. The memory access pattern consists of 12 stride 6 loads, and 2 stride 2 stores, and the underlying C type is `float`. GCC does not vectorize this program, and the approach of Nuzman et al. is not applicable.

The additional effect of applying reordering, reassociation, and blend merging is visible in many of the benchmarks, particularly `cxdotp − 2D`. Generally speaking, our simple canonical approach produces code that performs slightly worse than scalar code. However, for benchmarks dominated by computation, such as the `vnorm` programs, the overall speedup from vectorization is large, despite suboptimal data movement. For `vnorm − 3D` and `vnorm − 5D`, applying our simple canonical approach results in a program which is more than 2.5x faster than scalar code.[2] Applying our optimization techniques improves the speedup factor on data movement, bringing the overall speedup to more than 3x. In the `vnorm` programs, the interleaved access pattern is the principal impediment to vectorization. Once it is removed, significant performance gain is possible.

In each of the 4 cases where GCC applies the technique of Nuzman et al. — `cxmul`, `cxdotp − 2D`, `vdotp − 2D`, and `vnorm − 2D` — we generate programs which run faster on our SSE experimental platform, with the exception of `vnorm − 2D` where the error bars overlap. This gain is primarily due to our optimization techniques, particularly reordering, which can eliminate permutation instructions from the program. The effect of merging blends is particularly visible in `cxdotp − 2D`. The program contains 16 mergeable blend operations, each of which has two inactive lanes. Blend merging reduces this to 8 blend operations where every lane is active. Even though blend merging does not change the asymptotic complexity of the generated code, it can lead to significant speedups in practice.

Real-World Benchmarking (AVX2)

Figure 13 presents the results of real-world benchmarking on our AVX2 experimental platform. The figure demonstrates that our approach yields portable performance improvements across these two platforms. For most of the benchmarks, doubling the

[2]GCC requires the *-ffast-math* option to vectorize the computation in the `vnorm` benchmarks, which contains a floating point square root operation. GCC generates a reciprocal square root operation followed by some iterations of the Newton-Raphson method for approximation of square roots. In order to fairly represent the effect of our techniques, we precisely duplicated this instruction selection.

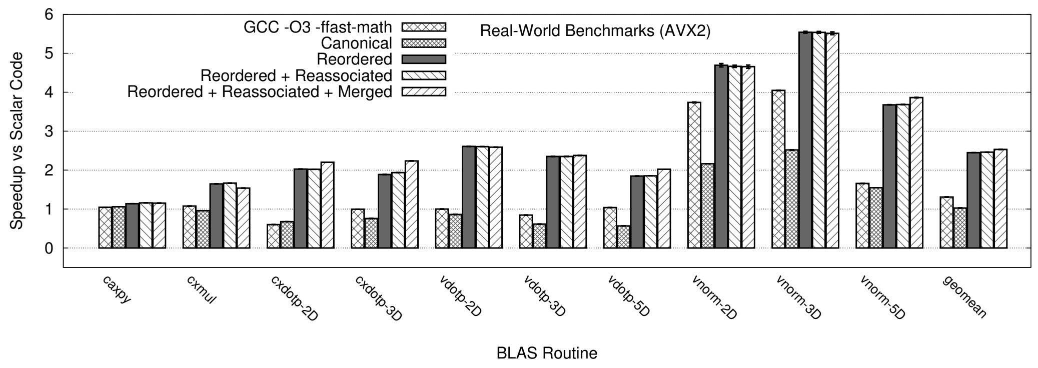


Fig. 13: **AVX2**: BLAS Level 1 operations exhibiting interleaved memory access optimized by GCC and using our approaches. Plot parameters are the same as for the SSE graph (Figure 12).

vectorization factor by moving from 128-bit to 256-bit vectorization yields a significant performance improvement. However, the benchmark caxpy is an exception. Our cost heuristic for instruction selection (Section 4.3) does not take into account fine-grained microarchitectural differences between the Haswell and Sandy Bridge platforms. Several 256-bit AVX2 versions of 128-bit SSE data reorganization instructions have either longer latency or require exclusive access to functional units where the SSE instruction does not. In addition, AVX2 instructions which reorganize data across the 128-bit boundary in a 256-bit register are subject to performance penalties relative to instructions which do not. In order to account for these differences, a detailed cost model would be required for instruction selection. However, for 9 out of 10 benchmarks, our technique results in efficient native AVX2 code. The caxpy benchmark exhibits a performance decrease relative to SSE in part because the instruction count is small, magnifying the effect of architectural differences.

Another significant difference from the SSE results is that the performance of the code generated by GCC is typically worse — GCC's code generation for AVX2 is not as mature as for SSE. On our SSE experimental platform, GCC achieves a geometric mean speedup of 1.43x over scalar code on this set of benchmark programs, but on AVX2 this is reduced to 1.30x. However, our approach achieves very good performance portability, represented by an increase in geometric mean speedup from a maximum of 1.77x on SSE to 2.53x on AVX2.

Comparison with Hand-Tuned and Reference BLAS

We performed some benchmarking of our generated code versus an open source reference BLAS implementation and Intel's MKL. The results are presented in Figure 14. For small problem sizes (in the range of 1K to 64K data elements) which exhibit dense interleaved data access, the code generated by our approach significantly outperforms both BLAS implementations experimented with. Single-core execution was used throughout. The performance gap begins to close only when the working set size grows so large that cache and memory effects come into play, i.e. at sizes which are ill-suited for single-core SIMD execution. Our approach could be used to produce optimized vector code for the individual single-core portions of a larger multi-core BLAS operation when the data access is interleaved.

Typically, BLAS implementations are tuned to take advantage of multicore parallelism and the memory hierarchy to achieve good performance when dealing with large amounts of data [Wang et al. 2014]. While BLAS implementations deal with both sparse and dense data representations, non-stride-1 (interleaved) dense data ac-

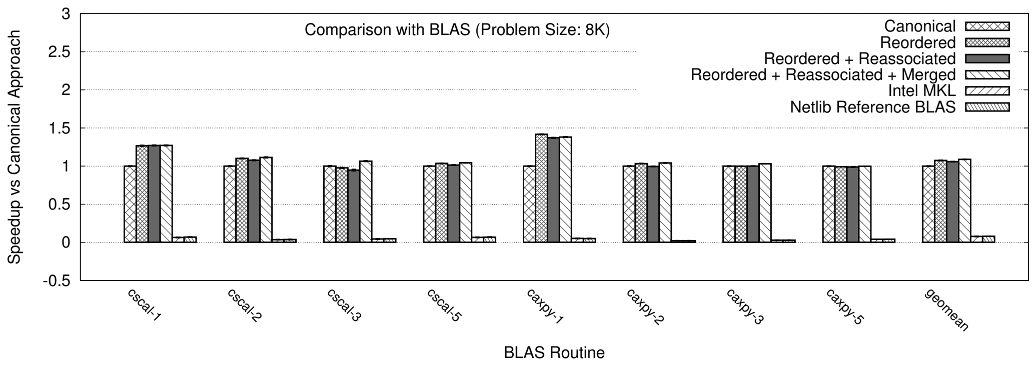


Fig. 14: **SSE**: BLAS Level 1 operations `axpy` and `scal` with interleaved memory access. The number in x-axis labels indicates the value of the *increment* parameter in the BLAS operation. Problem size in the graph title indicates the length of arrays. BLAS operation type is `complex`.

cess is difficult to optimize in a library context. It would be possible to provide a small number of hand-tuned kernels specialized for common strides, but doing so for every possible stride is implausible. Automated tuning systems such as ATLAS [Whaley et al. 2001] perform install-time code generation to automatically create an optimized library for the target platform. To avoid the need to generate an optimized kernel for every possible stride, a complementary compile-time specialization of operations using an approach similar to SPIRAL [Franchetti and Puschel 1993] could be used. This seems like a promising direction for future work.

5. RELATED WORK

5.1. SPIRAL

The initial version of the SPIRAL system [Franchetti and Puschel 1993] vectorized certain classes of interleaved memory access by translation to C macros. A set of handwritten implementations of these macros using SIMD intrinsics was included for each target platform. However, the class of interleaved access vectorized by SPIRAL is distinct from that vectorized by our approach, which is specifically targeted at affine interleaving (Definition 1). Subsequently the authors proposed an approach to automatic generation of vectorized code for their class of interleaved memory access [Franchetti and Püschel 2008], but their approach differs from ours in two key respects. The authors show that for the class of data permutations they consider, their technique produces a locally optimal code sequence for any one data permutation. However, locally optimal treatment of each individual permutation does not guarantee a global optimum for multiple permutations. In fact, both our work and the work of Nuzman et al. [2006] have shown that optimizing multiple simultaneous permutations with spatial locality as a group allows further optimization and sharing of overheads. Furthermore, the approach proposed [Franchetti and Püschel 2008] considers only shuffling operations. A key innovation of our approach is the decomposition of the problem of vectorizing interleaved access into separate permutation and blending phases. We have shown in Section 3 that this leads to the amortization of overhead across multiple interleaved accesses, and exposes fine-grained opportunities for optimizations like our blend merging technique (Section 3.4) which uses a single instruction stream to perform multiple interleaving operations in a parallel, shared-register fashion. In addition, the use of blending operations can lead to the elimination of permutations entirely (Section 3.1).

5.2. SLP

Vectorization techniques based on superword-level parallelism (SLP) [Larsen and Amarasinghe 2000] incorporate a tactic which can handle a restricted case of interleaved memory access. SLP attempts to transform programs so that interleaved memory accesses become consecutive, by searching for an unrolling factor for scalar loops which results in a dense memory access pattern in the vectorized loop. Such an unrolling factor can only be found if the scalar loop touches every element of each accessed array region within a fixed number of loop iterations, that is, if the access pattern of the loop has no gaps.

Previous work on SLP [Shin et al. 2005] has incorporated blending operations. The work extends SLP to enable it to vectorize control flow by converting conditional expressions in the scalar loop into predicated vector expressions (if-conversion), making use of blending operations to represent predicated results. However, we apply blending operations in a very different way, using them to perform interleaving/deinterleaving.

An recent extension to SLP [Liu et al. 2012] uses the polyhedral model [Cohen et al. 2004] to generate a non-SIMD data movement phase which gathers nonconsecutive memory elements in compact temporary arrays in the prologue of the vectorized loop, allowing computations within the loop to access them as though they were consecutive in memory. This approach sidesteps the need to generate vectorized code to perform data movement, and is sufficient to achieve speedup over scalar code for some programs. However, the approach assumes read-only array references, does not attempt to deal with interleaved writes, and requires accessed data to be copied to a temporary array before every loop iteration. Our approach synthesizes vectorized code to perform interleaved reads and writes in the vectorized loop, where instruction-level parallelism between data movement and computation can offset the overhead of memory access.

5.3. Loop, Function, and Whole-Program Vectorization

A key difference with much existing work is that we do not take a fixed permutation and try to generate fewest instructions to perform it. Rather, our techniques synthesize SIMD instruction sequences to perform interleaved memory access, which may contain permutations. The most closely related work, that of Nuzman et al. [2006] is discussed in depth in Section 4.2. Kudriavtsev and Kogge [2005] propose to reorder operations within SIMD instructions to minimize the number of permutations in the program. Using their vectorization approach, permutations can occur when multiple scalar operations read a common subexpression, or as a result of permutation in the source program. While the aim of minimizing permutations is similar to the aim of our reordering approach (Section 3.1) the key difference is that Kudriavtsev and Kogge require memory access to be consecutive, and reorder operations to minimize permutations resulting from computation. We reorder operations specifically to minimize permutations resulting from interleaving/deinterleaving. Future work could consider a combined approach, but the resulting multi-objective optimization problem appears hard.

Ren et al. [2006] optimize straight-line code by merging, propagating, and decomposing permutations within a basic block. Although their work does not address vectorization of interleaved memory access, they note that it often introduces permutations, using a power-of-two stride example which can be vectorized by Nuzman et al. [2006]. They further note that producing optimal code (that is, with fewest permutations) for an arbitrary basic block maps to the NP-hard multiterminal cut problem, and propose a practical heuristic solution. The approach of Ren et al. could be applied to further optimize the permutations in our synthesized programs.

Karrenberg and Hack [2011] propose a holistic approach to vectorization of whole functions, encompassing control and data flow. However, their approach assumes consecutive memory access. If combined with the approach to vectorizing interleaved access which we propose, that restriction would be lifted for programs where the stride of access is known at compile time, enabling broader application of their approach.

Park et al. [2012] propose a “SIMD Defragmentation” approach which tries to extract parallelism at the level of subgraphs of operations within the program, similar to SLP. Where vectorization using their approach results in interleaved memory accesses, our techniques could be applied to synthesize optimized SIMD code sequences to perform the access.

Barik et al. [2010] propose an approach for efficient selection of vector instructions for straight-line code sections, which is tightly integrated with register allocation and instruction scheduling. Although their cost model formulation includes parameters for the cost of packing or unpacking data in vector registers, they do not propose a technique for generating the code which performs interleaved access. Their experimental evaluation compares their approach to a prototype of SLP [Larsen and Amarasinghe 2000] using benchmark programs with restricted memory access patterns of the type discussed in Section 5.2. However, our techniques and their optimization approach are synergistic — if both were available in the compiler, their cost model could be extended to incorporate the costs of interleaved access vectorized using our approach. Similar to the work of Karrenberg and Hack [2011], this would enable broader application of both techniques.

Eichenberger et al. [2004], propose an approach to solve alignment issues while vectorizing. However, the focus of that work is reducing the cost of misaligned vector accesses resulting from unit-stride code. The operation of the initial permutation phase in our approach is similar to realignment using the dominant shift policy of Eichenberger et al., but our objective is not to reduce the cost of realignment, but of interleaving/deinterleaving. Our approach locally misaligns accesses within a set of vector registers acting as a compiler-controlled cache, reducing the number of instructions required for interleaving/deinterleaving. However, because accessed data is cached in vector registers, the misalignment does not translate into misaligned memory accesses.

6. CONCLUSION AND FUTURE WORK

We revisited the problem of automatic vectorization of interleaved memory access. Our generalized approach builds on the approach of Nuzman et al. to achieve significant speedup on programs which previously have been considered to require specialized, irregular, or hand-tailored solutions. Our approach vectorizes interleaved access patterns with arbitrary compile-time constant strides and gaps, two common impediments to automatic vectorization with existing techniques. In combination with the novel program transformations we propose, our vectorization approach results in mean speedups over scalar code of 1.77x (SSE) and 2.53x (AVX2) in real-world benchmarking on a selection of BLAS Level 1 routines, versus improvements of 1.43x (SSE) and 1.30x (AVX2) attained by GCC 5.0.

Possible extensions to the presented approach include relaxing the constraint on equivalent unit sizes in an access group (Section 2). This would enable our approach when dealing with complex array-of-structures memory layouts where adjacent structure fields are of different widths. Analysis and code generation for this use case appears significantly more involved, but it seems plausible that performance gain in this scenario is possible. Future work could also involve generalizing the optimizations presented to enable them to be applied as standalone utilities in other contexts. In addition, compile-time or run-time specialization of BLAS Level 1 operations exhibiting interleaved access seems like a promising direction for future work.

REFERENCES

Alfred V Aho, Mahadevan Ganapathi, and Steven WK Tjiang. 1989. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 11, 4 (1989), 491–516.

A Balachandran, Dhananjay M. Dhamdhere, and S Biswas. 1990. Efficient retargetable code generation using bottom-up tree pattern matching. *Computer Languages* 15, 3 (1990), 127–140.

Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. 2010. Efficient Selection of Vector Instructions Using Dynamic Programming. In *MICRO*. 201–212.

Albert Cohen, Sylvain Girbal, and Olivier Temam. 2004. A Polyhedral Approach to Ease the Composition of Program Transformations. In *Euro-Par*. 292–303.

Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. 2004. Vectorization for SIMD architectures with alignment constraints. In *PLDI*. 82–93.

Liza Fireman, Erez Petrank, and Ayal Zaks. 2007. New algorithms for SIMD alignment. In *Compiler Construction*. Springer, 1–15.

Franz Franchetti and Markus Puschel. 1993. A SIMD vectorizing compiler for digital signal processing algorithms. In *Vehicle Navigation and Information Systems Conference, 1993., Proceedings of the IEEE-IEE*. IEEE, 7–pp.

Franz Franchetti and Markus Püschel. 2008. Generating SIMD Vectorized Permutations. In *CC*. 116–131.

Christopher W Fraser, Robert R Henry, and Todd A Proebsting. 1992. BURG: fast optimal instruction selection and tree parsing. *ACM Sigplan Notices* 27, 4 (1992), 68–76.

Ralf Karrenberg and Sebastian Hack. 2011. Whole-function vectorization. In *CGO*. 141–150.

Alexei Kudriavtsev and Peter Kogge. 2005. Generation of permutations for SIMD processors. *SIGPLAN Not.* 40, 7 (June 2005), 147–156. DOI:http://dx.doi.org/10.1145/1070891.1065931

Samuel Larsen and Saman P. Amarasinghe. 2000. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*. 145–156.

Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)* 5, 3 (1979), 308–323.

Ruby B Lee. 1995. Accelerating multimedia with enhanced microprocessors. *IEEE Micro* 15, 2 (1995), 22–32.

Jun Liu, Yuanrui Zhang, Ohyoung Jang, Wei Ding, and Mahmut T. Kandemir. 2012. A compiler framework for extracting superword level parallelism. In *PLDI*. 347–358.

Saeed Maleki, Yaoqing Gao, María Jesús Garzarán, Tommy Wong, and David A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *PACT*. 372–382.

Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-vectorization of interleaved data for SIMD. In *PLDI*. 132–143.

Gabriele Paoloni. 2010. How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures. *Intel Corporation, September* (2010).

Yongjun Park, Sangwon Seo, Hyunchul Park, Hyoun Kyu Cho, and Scott Mahlke. 2012. SIMD defragmenter: efficient ILP realization on data-parallel architectures. In *ACM SIGARCH Computer Architecture News*, Vol. 40. ACM, 363–374.

Gang Ren, Peng Wu, and David Padua. 2006. Optimizing data permutations for SIMD devices. In *ACM SIGPLAN Notices*, Vol. 41. ACM, 118–131.

Kenneth Rosen. 2011. *Discrete Mathematics and Its Applications 7th edition*. McGraw-Hill.

Thomas Schaub, Simon Moll, Ralf Karrenberg, and Sebastian Hack. 2015. The impact of the SIMD width on control-flow and memory divergence. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 4 (2015), 54.

Jaewook Shin, Jacqueline Chame, and Mary W. Hall. 2002. Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures. In *IEEE PACT*. 45–55.

Jaewook Shin, Mary Hall, and Jacqueline Chame. 2005. Superword-level parallelism in the presence of control flow. In *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 165–175.

Deependra Talla, Lizy Kurian John, and Doug Burger. 2003. Bottlenecks in multimedia processing with SIMD style extensions and architectural enhancements. *Computers, IEEE Transactions on* 52, 8 (2003), 1015–1031.

Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel Math Kernel Library. In *High-Performance Computing on the Intel® Xeon Phi*. Springer, 167–188.

R Clint Whaley, Antoine Petitet, and Jack J Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 1 (2001), 3–35.

Michael Joseph Wolfe. 1996. *High performance compilers for parallel computing*. Addison-Wesley.

A. ALGORITHMS

ALGORITHM 1: Generate code to vectorize a read access (canonical)

```
Input: A strided access a, load-mapped register set p
Output: s[VF − 1] contains VF packed consecutive elements of a
$s \leftarrow$ allocate VF temporary registers;
for $i = 0$ to VF − 1 do
    mask ← ⟨VF × ∗⟩;
    mask[i] ← ((stride ∗ $i$) + offset) mod VF;
    r ← p[(stride ∗ $i$)/VF];
    generate: permute r, mask, s[i];
end
for $j = 1$ to VF − 1 do
    left ← $j$, right ← VF − $j$;
    mask ← ⟨(left × L) + (right × R)⟩;
    generate: blend s[j − 1], s[j], mask, s[j];
end
```

ALGORITHM 2: Generate code to vectorize a write access (canonical)

```
Input: VF consecutive elements of a strided access a packed in register r
Output: Store-mapped register set p contains the VF elements of a in store order
pmask ← ⟨VF × ∗⟩;
bmask ← ⟨VF × L⟩;
pmasks ← stride copies of pmask;
bmasks ← stride copies of bmask;
$s \leftarrow$ allocate stride temporary registers;
for $i = 0$ to VF − 1 do
    register ← (stride ∗ $i$)/VF;
    lane ← (stride ∗ $i$) mod VF;
    pmasks[register][lane] ← $i$;
    bmasks[register][lane] ← R;
    generate: permute r, pmasks[register], s[register];
    generate: blend p[register], s[register], bmasks[register], p[register];
end
```

ALGORITHM 3: Generate code to deinterleave a strided read without permutation

```
Input: A strided access a, load-mapped register set p, a collision free at VF
Output: Register s contains VF packed consecutive elements of a (out-of-order)
mask ← ⟨VF × $L$⟩;
masks ← stride copies of mask;
for $i = 0$ to VF − 1 do
    maskIdx ← ((stride ∗ $i$) + offset) mod VF;
    laneIdx ← ((stride ∗ $i$) + offset) / VF;
    masks[maskIdx][laneIdx] ← $R$;
end
for $j = 1$ to VF − 1 do
    generate: blend s, p[j − 1], masks[j − 1], s;
end
```

ALGORITHM 4: Generate code to interleave a strided write without permutation

```
Input: VF consecutive elements of a strided access a packed in register r, a collision free at VF
Output: Store-mapped register set p contains the VF elements of a (in store order)
mask ← $\langle \mathtt{VF} \times L \rangle$;
masks ← stride copies of mask;
for $i = 0$ to $\mathtt{VF} - 1$ do
    maskIdx ← $((\mathtt{stride} * i) + \mathtt{offset}) \mod \mathtt{VF}$;
    laneIdx ← $((\mathtt{stride} * i) + \mathtt{offset}) \ / \ \mathtt{VF}$;
    masks[maskIdx][laneIdx] ← $R$;
end
for $j = 1$ to $\mathtt{VF} - 1$ do
    generate: blend p[j − 1], r, masks[j − 1], p[j − 1];
end
```

ALGORITHM 5: Generate code to vectorize a group of shared-stride reads with lane collisions

```
Input: Load-mapped register set p, rotations for each mapped register.
Output: Compute register set c produced, with some registers out-of-order
span ← stride ∗ VF;
lanes ← VF;
vectors ← span/lanes;
for $i = 0$ to vectors − 1 do
    mask ← $\{(x + \mathtt{rotations[i]}) \mod \mathtt{lanes} \mid x \leftarrow [0..(\mathtt{lanes} - 1)]\}$;
    generate: permute p[i], mask, p[i];
end
foreach a in accesses do
    $\mathtt{c_a} \leftarrow \mathtt{p_0}$;
    for $j = 1$ to lanes − 1 do
        left ← $j$, right ← lanes − $j$;
        mask ← $rotate(\mathtt{rotations}[j], \langle(\mathtt{left} \times \mathtt{L}) + (\mathtt{right} \times \mathtt{R})\rangle)$;
        generate: blend $\mathtt{c_a}$, p[j], mask, $\mathtt{c_a}$;
    end
end
```

ALGORITHM 6: Generate code to vectorize a group of shared-stride writes with lane collisions

```
Input: Register set c with n packed shared-stride accesses, with a common SIMD lane order
Output: Store-mapped register set p contains all accesses in rotated store order
span ← stride ∗ VF;
lanes ← VF;
vectors ← span/lanes;
foreach a in accesses do
    for $i = 0$ to vectors − 1 do
        mask ← $\langle \mathtt{lanes} \times \mathtt{L} \rangle$;
        mask[$(\mathtt{offset_a} + \mathtt{i}) \mod \mathtt{lanes}$] ← R;
        generate: blend p[i], $\mathtt{c_a}$, mask, p[i];
    end
end
for $i = 0$ to $n - 1$ do
    mask ← $\{(x + (\mathtt{lanes} - i)) \mod \mathtt{lanes} \mid x \leftarrow [0..(\mathtt{lanes} - 1)]\}$;
    generate: permute p[i], mask, p[i];
end
```