Auto-Vectorization of Interleaved Data for SIMD

Dorit Nuzman Ira Rosen Ayal Zaks

IBM Haifa Labs - HiPEAC Member, Haifa University Campus, Mount Carmel, Haifa 31905, Israel
{dorit,irar,zaks}@il.ibm.com

Abstract

Most implementations of the Single Instruction Multiple Data (SIMD) model available today require that data elements be packed in vector registers. Operations on disjoint vector elements are not supported directly and require explicit data reorganization manipulations. Computations on non-contiguous and especially interleaved data appear in important applications, which can greatly benefit from SIMD instructions once the data is reorganized properly. Vectorizing such computations efficiently is therefore an ambitious challenge for both programmers and vectorizing compilers. We demonstrate an automatic compilation scheme that supports effective vectorization in the presence of interleaved data with constant strides that are powers of 2, facilitating data reorganization. We demonstrate how our vectorization scheme applies to dominant SIMD architectures, and present experimental results on a wide range of key kernels, showing speedups in execution time up to 3.7 for interleaving levels (stride) as high as 8.

Categories and Subject Descriptors D.3.4 [*Processors*]: compilers, optimization; C.1.1 [*Single Data Stream Architectures*]: RISC/CISC, VLIW architectures; C.1.2 [*Multiple Data Stream Architectures (Multiprocessors)*]: Single-instruction-stream, multiple-data-stream processors (SIMD)

General Terms Performance, Algorithms

Keywords SIMD, vectorization, subword parallelism, data reuse, Viterbi

1. Introduction

In recent years, a wide range of modern computational platforms have incorporated Single Instruction Multiple Data (SIMD) extensions into their processors. SIMD capabilities are now common in modern digital signal processors [7, 14], gaming machines [15], general purpose processors [25, 8], and massively parallel supercomputers [3]. These SIMD mechanisms allow architectures to exploit the natural parallelism present in applications from different domains, including signal processing, games, and scientific codes, by simultaneously executing the same instruction on multiple data elements.

Optimizing compilers use vectorization techniques [38] to exploit the SIMD capabilities of these architectures. Such techniques reveal data parallelism in the scalar source code and transform groups of scalar instructions into vector instructions. However, it is often complicated to apply vectorization techniques to SIMD architectures [30] because these architectures are largely non-uniform, supporting specialized functionalities and a limited set of data types. Vectorization is often further impeded by the SIMD memory architecture, which typically provides access to contiguous memory items only, often with additional alignment restrictions. Computations, on the other hand, may access data elements in an order that is neither contiguous nor adequately aligned. Bridging this gap efficiently requires careful use of special mechanisms including permute, pack/unpack, and other instructions that incur additional performance penalties and complexity. These mechanisms differ widely from one SIMD platform to another. A vectorizing compiler must be aware of these capabilities with their associated costs and use them appropriately to produce high quality code.

In this paper, we focus on automatic vectorization of computations that require data-reordering techniques, using an optimizing compiler for SIMD targets. We show how our scheme solves data-reordering problems efficiently while exploiting data reuse, thereby enabling the vectorization and acceleration of a wide range of patterns. Furthermore, our vectorization scheme is implemented in a generic and multi-platform setting. To the best of our knowledge, this is the first solution to the data-reordering problem in the context of an optimizing compiler for a variety of SIMD targets.

The contributions of this paper are as follows:

- New generic vectorization technique for interleaved data that efficiently vectorizes non-contiguous access patterns with constant strides that are power of 2, while exploiting spatial reuse.
- Demonstration of our technique's applicability on a wide range of real-world kernels, exhibiting a range of access patterns, with strides from 2 to 8.
- Extensive qualitative and experimental evaluation that show the compiler can achieve speedups for a range of interleaving factors (strides), as high as 32.

The paper is organized as follows: Section 2 presents the data interleaving problem, the challenges it raises, and the abstractions we introduce to express the unique data reordering features of SIMD platforms. In Section 3, we give a brief overview of the GCC compiler used and its vectorizer infrastructure. Section 4 describes our extensions to the vectorizer to handle interleaved data. Qualitative analysis of the potential overheads associated with vectorizing interleaved accesses are presented in Section 5, along with experimental results for synthetic tests. In Section 6 we provide experimental results obtained by running the vectorizer on real-world kernels. Section 7 brings up an interesting connection between our work and SLP. Section 8 discusses related work and Section 9 concludes.

PLDI'06 June 10–16, 2006, Ottawa, Ontario, Canada

```
for(int i = 0; i < len; i++){
    c[2i] = a[2i]*b[2i] - a[2i+1]*b[2i+1];
    c[2i+1] = a[2i]*b[2i+1] + a[2i+1]*b[2i];
}
```

Figure 1. Scalar complex multiplication

```
for(int i = 0; i < len; i+=VF){
    vector abee = (a[2i] * b[2i], a[2i+2] * b[2i+2], ...,
                   a[2i+2(VF-1)] * b[2i+2(VF-1)];
    vector aboo = (a[2i+1] * b[2i+1], a[2i+3] * b[2i+3], ...,
                   a[2i+2(VF-1)+1] * b[2i+2(VF-1)+1];
    vector abeo = (a[2i] * b[2i+1], a[2i+2] * b[2i+3], ...,
                   a[2i+2(VF-1)]*b[2i+2(VF-1)+1];
    vector aboe = (a[2i+1] * b[2i], a[2i+3] * b[2i+2], ...,
                   a[2i+2(VF-1)+1]*b[2i+2(VF-1)];
    c[2i,2i+2,...,2i+2(VF-1)] = abee - aboo;
    c[2i+1,2i+3,...,2i+2(VF-1)+1] = abeo + aboe;
}
```

Figure 2. Vectorized complex multiplication

2. The Interleaving Problem

Vector instructions of conventional SIMD extensions access multiple data that is adequately packed in vector registers. In contrast, computations may execute the same operation on multiple data that is placed in non-consecutive or arbitrarily ordered locations. For example, consider the complex multiplication of two arrays having their real and imaginary elements interleaved, as depicted in Figure 1.

Using a standard loop-based vectorization scheme for SIMD, multiple occurrences of the same operation across consecutive iterations can be grouped into single SIMD instructions, as shown in Figure 2, where `VF` is the vectorization factor — the number of elements that fit in a vector register. (We assume for clarity that `len` is divisible by `VF`.)

The vector multiplications in the example require that the even and odd elements of arrays `a` and `b` be loaded from memory and packed in vector registers. However, most SIMD architectures allow the loading and storing of multiple data from memory only if the addresses $a(i)$ are consecutive, i.e., of the form $a(i) = b + ui$, often requiring the base address b to be properly aligned. Here, u is the unit size in bytes of a single element, where $\texttt{VS} = u \cdot \texttt{VF}$ is the vector size in bytes (as illustrated in Figure 4), and $i = 0, 1, \ldots, \texttt{VF} - 1$.

The restriction to load and store from consecutive memory addresses implies that in order to access the odd or even elements separately, they need to be extracted after loading them both consecutively into vector registers. A similar case applies to storing elements in non-consecutive addresses (see Figure 3).

The example in Figure 3 demonstrates the use of several `extract` and `interleave` operations that copy `VF` elements from two given vector registers forming another register with the desired elements (in the desired order) in order to cope with loading from and storing to addresses of the form $a(i) = b + 2ui$ (see Figure 5). Such access patterns are generally referred to as *non-unit stride* accesses, of the form $a(i) = b + \delta ui$ for some constant integer $\delta \neq 1$, called the *interleaving factor* or *stride*. As we see later, these `extract` and `interleave` abstractions play an important role in handling strided accesses by the compiler.

Each scalar iteration can have n different memory accesses in the range of δ consecutive elements. When $n = \delta$, the access pattern is *fully interleaved*, and when $n < \delta$ the interleaved

```
for(int i = 0; i < len; i+=VF){
    vector a1 = load(a[2i],a[2i+1],...,a[2i+VF-1]);
    vector a2 = load(a[2i+VF],a[2i+VF+1],...,a[2i+2VF-1]);
    vector ao = extract_odds(a1,a2);
    vector ae = extract_evens(a1,a2);
    vector b1 = load(b[2i],b[2i+1],...,b[2i+VF-1]);
    vector b2 = load(b[2i+VF],b[2i+VF+1],...,b[2i+2VF-1]);
    vector bo = extract_odds(b1,b2);
    vector be = extract_evens(b1,b2);
    vector abee = ae * be;
    vector aboo = ao * bo;
    vector abeo = ae * bo;
    vector aboe = ao * be;
    ce = abee - aboo;
    co = abeo + aboe;
    c[2i,2i+1,...,2i+VF-1] = interleave_low(ce,co);
    c[2i+VF,2i+VF+1,...,2i+2VF-1] = interleave_high(ce,co);
}
```

Figure 3. Vectorized complex multiplication with data reordering

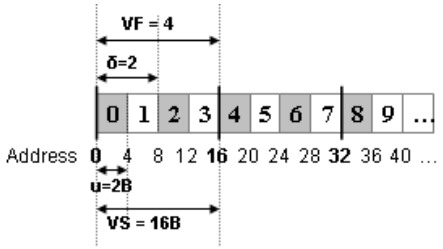

Figure 4. Memory access pattern of the code example

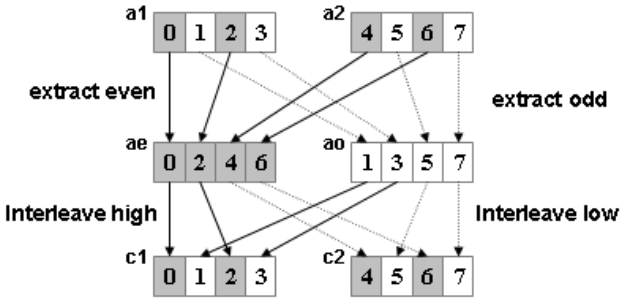

Figure 5. Extract_even/odd and interleave_low/high operations

access-pattern contains *gaps* (in our code example in Figure 1, $n = \delta = 2$). The classic approach to vectorization tries to operate on `VF` elements in each vector iteration. Vectorizing a strided access with interleaving factor δ would therefore require the load/store of δ`VF` consecutive (and aligned) elements, from which the desired `VF` elements for the original access are extracted/inserted. (A strided access having $\delta \geq \texttt{VF}$ can load/store only `VF` non-consecutive vectors, but we always consider the full interval of δ consecutive vectors to facilitate spatial reuse of several interleaved strided accesses.) Several (n) vector accesses can extract their elements from the set of δ`VF` consecutive elements; it is important to recognize interleaved (and in particular, fully interleaved) accesses to exploit spatial locality and also to facilitate stores that typically cannot tolerate gaps.

In this paper we address the interleaving problem — the automatic recognition of interleaved accesses and generation of opti-

mized code to efficiently vectorize them. We also use the general scheme of loading additional elements and then extracting the desired elements to solve the ubiquitous alignment problem [9, 39]. Our treatment of interleaved accesses also takes care of alignment considerations.

The interleaving problem is a special case of the general 'scatter-gather' notion, which supports arbitrary access patterns using indirection tables. In general, data reordering is required when the input to one vector operation is a permutation of another vector operation's output, or a combination of outputs produced by several vector operations (including vector loads). In such cases the data from one or more vector registers needs to be permuted or extracted before it can be fed to the corresponding SIMD instruction. We restrict our attention to strided accesses because their overhead can be reduced as compared to less regular or arbitrary accesses.

A major factor that defines the nature of the solution to the problem of vectorization of interleaved data is the vector platform being targeted. Different models of vector machines offer different architectural mechanisms to deal with non-consecutive data, and pose different challenges to solving this problem. One example is vector machines that support 'scatter-gather' operations in hardware. Another example of unique architectural support for non-consecutive data is the eLite DSP architecture that supports SIMdD operations (Single Instructions on Multiple disjoint Data), via vector pointers, in which data reordering bears a smaller overhead.

Our focus is on existing SIMD architectures/extensions, such as Altivec [8] and MMX/SSE [25, 34]. In recent years, a variety of implementations of the SIMD model have been incorporated into many platforms, most of which offer special mechanisms to shuffle data between vector registers [2, 8, 23, 25, 32]. These mechanisms incur an overhead when applied repeatedly (e.g., within a loop), and reducing this overhead of data reordering can be critical to performance.

The most general SIMD mechanism available to handle interleaved data is the `permute` operation, such as the Altivec `vperm` instruction. It takes two vectors as input and treats them as one stream of elements. The third argument is a permutation mask that, for each of the elements in the output vector, specifies the index of an element from the input stream to be placed at the corresponding location in the output vector. In our example in Figure 5, a `permute` operation with mask (0,2,4,6) extracts the elements at these indices from the concatenated vector $a1||a2$. On most platforms with SIMD support (MMX, SSE, MIPS, IA-64 and SPE), this permute operation is not supported in its generic form [22]. It usually requires immediate values to the permutation mask, or allows only a fixed permutation or a permutation limited to extracting exactly half the elements from one vector and the other half from the second vector (e.g., SSE's `shufps`). Only the AltiVec `vperm` instruction allows arbitrary, variable permutation. For this reason, we introduced the `extract_even` and `extract_odd` abstractions (Figure 5) rather than a generic shuffle/permute idiom. Other reasons for this choice are revealed in Section 4.2.

3. Vectorizer Overview

We used GCC (the GNU Compiler Collection [12]) of the Free Software Foundation, to implement the vectorization of computations that involve interleaved data. In this section, we describe the infrastructure of GCC that is relevant to our work. In the next section we indicate how this infrastructure was extended to support vectorization of interleaved accesses.

GCC uses multiple levels of Intermediate Languages (IL) in the course of translating the original source language to assembly. Our focus is on GIMPLE [19], a new IL that supports Static Single Assignment (SSA) [24] and retains enough information from the source language to facilitate advanced data-dependence analysis and aggressive high-level optimizations, including auto-vectorization [11]. From the high-level target-independent GIMPLE IL, statements are translated to the low-level instructions of the RTL (Register Transfer Language), where target-dependent optimizations, such as instruction scheduling and register allocation, are applied.

```
vect_analyze_loop (struct loop *loop) {
  loop_vec_info loopinfo;
  loop_vinfo = vect_analyze_loop_form (loop);
  if (!loop_vinfo) FAIL;
  if (!determine_VF (loopinfo)) FAIL;
  if (!analyze_data_refs (loopinfo)) FAIL;
  if (!analyze_scalar_cycles (loopinfo)) FAIL;
  if (!analyze_data_ref_dependences (loopinfo)) FAIL;
  if (!analyze_data_ref_accesses (loopinfo)) FAIL;
  if (!analyze_data_refs_alignment (loopinfo)) FAIL;
  if (!analyze_operations (loopinfo)) FAIL;
  LOOP_VINFO_VECTORIZABLE_P (loopinfo) = 1;
  return loopinfo;
}

vect_transform_loop (struct loop *loop) {
  FOR_ALL_STMTS_IN_LOOP(loop, stmt)
    vect_transform_stmt (stmt);
  vect_transform_loop_bound (loop);
}
```

Figure 6. Vectorizer outline

Figure 6 outlines the GCC vectorization pass, which serves as the basis for incorporating our support for interleaved data access. The vectorizer follows the classic theory of loop-based, dependence-base vectorization [1, 13, 38], and applies a set of eight analyses to each loop (see `vect_analyze_loop()`), followed by the actual vector transformation (`vect_transform_loop()`) for loops that successfully complete the analysis phase:

1. The first analysis phase, `vect_analyze_loop_form()` identifies innermost, single basic block countable loops. (Certain multiple basic block constructs may be collapsed into single basic blocks containing conditional operations by an if-conversion pass prior to vectorization). A loop is considered countable if the number of iterations can be determined prior to entering the loop, either as a compile-time constant or as a variable evaluated at runtime.
2. The `determine_VF()` phase scans all the operations in the loop and determines the vectorization factor `VF`, which represents the number of data elements to be packed together in a vector, and is also the strip-mining factor of the loop. The `VF` is determined according to the data-types that appear in the loop and the vector sizes supported by the target platforms. The GCC vectorizer currently supports a single vector size per platform.
3. Next, `analyze_data_refs()` finds all the memory references in the loop and checks if they are 'analyzable'. This means that an access function describing the series of addresses across iterations of the loop can be constructed using scalar evolution analysis [27, 28]. The access function is needed for memory-dependence, access-pattern, and alignment analyses.
4. Dependence cycles represent recurrences that must be handled to enable vectorization of a loop. The function `analyze_scalar_cycles()` examines dependence cycles that involve scalar variables (i.e., do not go through memory), such as reductions, and verifies that they can be handled appropriately.
5. The `analyze_data_ref_dependences()` phase checks that the dependence distance between every pair of data references in the loop is either zero (i.e., intra-loop dependence) or at least `VF`, by applying a set of classic data dependence tests [1, 13, 38].

Potentially aliased references are handled at this phase by using static alias analysis or by introducing dynamic checks. Pairs of instructions whose distance is a multiple of `VF` are marked as having the same alignment.

6. Next, `analyze_data_ref_accesses()` checks that every reference to memory accesses consecutively increasing addresses. This must be changed to support interleaved accesses, as shown in Section 4.
7. The `analyze_data_ref_alignment()` phase makes sure the alignment of all the data references in the loop can be supported, either by means of loop peeling or loop versioning to force the alignment of the data references, or by directly vectorizing the unaligned accesses using proper alignment handling idioms.
8. The final analysis phase `analyze_operations()` verifies that every operation in the loop can be supported in vector form by the target architecture.

The vectorization transformation can be generally described as *strip-mine by VF and substitute one-to-one*, which implies that each scalar operation in the loop is replaced by its vector counterpart. The loop transformation phase scans all the statements of the loop from the top down (vectorizing definitions before their uses) and inserts a vector statement in the loop for each scalar statement that needs to be vectorized. Often additional handling beyond the one-to-one substitution is provided to generate code before or after the loop: constants and loop invariants require that vectors be initialized at the loop pre-header; reduction and induction computations require special epilog code after the loop. In other cases, single scalar operations are replaced by more than one vector operation; such is the case for supporting access to unaligned or non-unit stride data, and also operations on mixed data types. Conversely, single vector operations may replace sequences of scalar operations by a special idiom recognition engine (e.g., saturating arithmetic).

In GIMPLE, vector operations are generally represented like scalar operations — the same operation codes are used, but the arguments are of vector types. This is suitable for 'pure SIMD' operations, in which the functionality represented by the operation code (e.g., addition) is performed on each element of the vector. Other vector operations such as reductions, alignment-support mechanisms, and vector element shuffling operations (e.g., `extract/interleave`), are meaningless in the context of scalar computations and therefore were not available in GIMPLE. In order to express these mechanisms in the vectorized GIMPLE IL, we introduced new platform-independent vector abstractions to GCC.

Finally, the loop bound is transformed to reflect the new number of iterations, and if necessary, an epilog scalar loop is created to handle cases of loop bounds that do not divide by the vectorization factor. (This epilog must also be generated in cases where the loop bound is not known at compile time.)

4. Extending the Vectorizer to Handle Interleaved Data

In this section, we describe the modifications we introduced to the `vect_analyze_loops()` and `vect_transform_loops()` phases of the vectorizer so it could handle interleaved accesses. This extension detects groups of instructions that access interleaved data with the same interleaving factor, and exploits their spatial locality.

4.1 Analyzing Interleaved Accesses

Our goal at the analysis stage is to partition the set of load and store instructions with non-unit stride access into groups, where each group contains useful spatial locality. These groups exhibit useful spatial locality and help estimate an accurate cost for supporting the interleaved accesses of the group. This cost helps to determine if vectorization is profitable. The cost model itself is outside the scope of this paper; here we focus on providing data for such a cost model, and vectorize any loop we can. After the analysis stage produces the above partition and determines that vectorization is feasible (and profitable), the transformation stage refers to each group and generates efficient vector code for its members.

A pair of loads or stores x, y with non-unit stride accesses of the form $a_x(i) = b_x + \delta_x u_x i$ and $a_y(i) = b_y + \delta_y u_y i$ exhibit useful spatial locality if they can repeatedly access elements from common preloaded/poststored vectors. Namely, if $\delta_x u_x = \delta_y u_y$ and $|b_x - b_y|$ is 'sufficiently' small. Most occurrences of interleaved accesses that exhibit spatial locality have the same unit sizes and strides

$$u = u_x = u_y \tag{1}$$

$$\delta = \delta_x = \delta_y \tag{2}$$

and can share the same δ vectors if relatively aligned:

$$|b_x - b_y| < \delta u. \tag{3}$$

We introduced a new data structure to represent a group of load (or store) instructions with non-unit stride accesses that exhibit pairwise useful spatial locality. All members of a group access the same array and have the same stride δ. Each member x of a group is assigned an integer j_x (possibly negative), called the *index*, such that $j_x - j_y = b_x - b_y$ for any two members x, y of a group. These indices help determine the relative reordering among the members of the group. Note that members of a group have distinct indices, due to a redundant load and store elimination pass that takes place prior to vectorization. When the number of members in a group is smaller than its stride, there are gaps. We currently restrict the number of members to be no larger than the stride, as this is the common case and simplifies the treatment.

The member with the smallest (if $\delta > 0$, or the largest if $\delta < 0$) index in a group is designated as the *leader* of the group. The leader is later responsible (at the transformation stage) for loading or storing the data for the group, and the base address b of the leader determines the overall starting address. All other members use the difference between their index and the leader's index to determine the extraction they need from the data provided by the leader. For each load and store instruction, we record a pointer to its group and a field for storing its index. These pointers enable group members to quickly navigation to their leader and their index.

A set of loads or stores can be assigned to the same group by analyzing them in pairs, following a greedy approach. We analyze pairs of loads and stores in an order that keeps one load or store fixed as a pivot; this way, groups are formed one by one and each group is augmented by one new member at a time, starting with the pivot. The natural place to accommodate this analysis is in the `analyze_data_ref_dependences()` phase, which already iterates over all pairs of loads and stores (in the desired pivoting order) to check for their dependence distance. We inserted the following logic there to detect pairs of loads and stores for grouping:

```
if (distance_between_accesses <= (VF-1)*stride*u)
  if (read,write) or (write,read) or (write,write)
    ok = dep_resolve();
  endif
endif

if ok and (distance_between_accesses < stride*u)
  if (read,read) or (write,write)
    ok = analyze_interleaving();
  endif
endif
```

The first check is the original dependence distance test to permit executing VF iterations in parallel, accommodating a non-unit stride δ. The second check is the relative alignment condition (3) for spatial reuse.

The analyze_interleaving function examines a pair of non-unit stride accesses for x, y that meet Conditions (1), (2) and (3), and builds interleaving groups incrementally. If both x and y have not yet been assigned to groups, we open one new group for the two accesses and assign them indices $j_x = 0$ and $j_y = j_x + b_y - b_x$. (Note that $|j_y - j_x| < \delta u$ due to Condition (3).) We also record the access with the smaller base address as the leader of the group and record the index of the other access as the largest index j_l. Otherwise, if one access, say that of x, has already been assigned to a group and the other (of y) has not, we set the index of y to be $j_y = j_x + b_y - b_x$. (Note that j_y can be negative; we do not change an index after it is set.) We compare j_y to the smallest index j_s (of the leader) and largest index j_l in the group of x: if $|j_y - j_s| \geq \delta u$ or $|j_y - j_l| \geq \delta u$, we keep y unassigned and continue to the next pair of loads or stores. Otherwise, if $|j_y - j_s| < \delta u$ and $|j_y - j_l| < \delta u$ we assign y to the group of x. We then check if j_y is smaller than j_s, and if so make y the new leader. We also update $j_l = \max(j_l, j_y)$. Finally, if both accesses have already been assigned to groups, we do not process them again. This relies on the pivoting order in which the pairs of accesses are processed and reduces the complexity of the grouping algorithm: we iterate over $O(n^2)$ pairs of loads and stores (where n is the number of loads and stores), spending $O(1)$ time with each pair and assigning a load or store to a group at-most once.

Referring to the example in Figure 1, three groups will be formed: a group for the loads from a[2i], a[2i+1], a group for the loads from b[2i], b[2i+1], and a group for the stores to c[2i], c[2i+1]. Note that duplicate loads (e.g., from a[2i]) have been eliminated leaving a single load. The strides of the groups are $\delta = 2 > 0$, therefore the leaders are the accesses with the smaller offset (to the even elements), and there are no gaps.

After all pairs of loads and stores have been traversed, we visit each group and estimate the profitability of vectorization, including the identification and assessment of gaps, which reduce the amount of reuse and may be intolerable for stores (see Section 5 for the profitability analysis). We introduced this scan over the groups into the current framework of the vectorizer at the analyze_data_ref_accesses() phase, which traverses each individual load and store. (Recall that this phase needs to be modified anyway to permit non-unit stride accesses.) Every time we reach a group leader during the analyze_data_ref_accesses() scan, we analyze its group. During this scan, we also look for strided accesses that were not assigned to any group; if found, we open and analyze new singleton groups for these accesses.

4.2 Extensions to Transformation Phase

We modified the transformation phase to vectorize interleaved data, based on the groups produced by the enhanced analysis phase. When reaching the first member of a group having stride δ, we generate δ load or store statements starting from the address of the leader. This amount of data will be loaded/stored in each iteration of the loop. If the base address of the leader is not known to be properly aligned, the standard treatment for (potentially) unaligned access is applied using *zero shift* policy [9] or loop peeling, see [22]. We then generate a complete set of $\delta \log_2 \delta$ data reordering statements of the form extract_even/odd (for loads) and interleave_low/high (for stores). These statements are capable of handling strides that are powers of 2. For example, the data accessed by the pattern $a(i) = b + 4i$, assuming b is aligned, can be extracted from four loads by a combination of extract_even statements, as depicted in Figure 7. The complete set of $\delta \log_2 \delta$

```
vector b1 = load(b,b+1,...,b+VF-1);
vector b2 = load(b+VF,b+VF+1,...,b+2VF-1);
vector b3 = load(b+2VF,b+2VF+1,...,b+3VF-1);
vector b4 = load(b+3VF,b+3VF+1,...,b+4VF-1);
vector b12e = extract_evens(b1,b2);
vector b34e = extract_evens(b3,b4);
vector b1234ee = extract_evens(b12e,b34e);
```

Figure 7. Extracting aligned stride-4 data using extract_evens

extract_even and extract_odd statements are generated after the vector loads at the location of the first scalar load. These statements are organized as $\log_2 \delta$ layers of δ statements each, where the last layer provides the required strided data for the possible offsets (e.g., b1234ee in Figure 7). Each member of the group is then connected to the appropriate resultant extract_odd/even statement of this last layer, according to its index relative to the index of the group leader. The extract_odd/even statements that remain unused, in the case of gaps, are discarded later by a dead-code elimination pass.

We handle stores to interleaved data analogously (only for gapless accesses, again for power-of-2 strides) by generating interleave_low/high statements instead of extract_odd/-even and placing them before the vector stores. These statements are placed next to the last scalar store. This completes the vectorizer extensions we introduced to handle interleaved data; no modifications were needed to handle the vectorization of non-load/store statements.

For strides that are not a power of 2, we need permute capabilities that are more flexible than extract_odd/even and interleave_low/high, which are provided by some platforms. In addition, each non-power-of-2 stride generally requires a tailored, irregular solution. On the other hand, the simple and generic extract_odd/even and interleave_low/high abstractions are supported efficiently on most SIMD extensions and are as good as specialized permutes for power-of-2 strides. This is because $\delta - 1$ operations are needed to extract the relevant data from δ vectors, even if each operation can extract an arbitrary set of VF elements from a pair of vectors.

We could have alternatively disregarded spatial reuse in the vectorizer by generating the required treatment for each instruction separately and relying on a subsequent redundant load/store elimination pass to later detect and exploit spatial reuse. However, it is better to have the vectorizer detect and handle such reuse opportunities for several reasons: exploiting such reuse greatly reduces the associated overhead that the vectorizer estimates to assess the profitability of vectorizing the loop; handling alignment complicates subsequent attempts to detect reuse opportunities; and storing non-consecutive data complicates the situation further, if it is at all permissible.

5. Experimental Results on Synthetic Tests

We evaluate the benefits of vectorizing computations with interleaved accesses by considering a wide range of interleaving factors and worst-case scenarios using a set of synthetic test-cases and kernels extracted from real-world benchmarks. The results we present in this and the next sections were generated automatically using the autovect-branch of the GCC 4.1 compiler, freely available from the FSF [12]. Experiments were performed on an IBM PowerPC 970 processor with Altivec support, running Linux. The PowerPC 970 is an out-of-order execution super-scalar processor with five scalar functional units (two fixed point, two floating point, and one branch), two SIMD units (one arithmetic, one permute) and two memory units shared between the scalar and vector units.

δ	VF=4	VF=8	VF=16
1	4	8	16
2	2	4	8
4	1.3	2.6	5.3
8	1	2	4
16	0.8	1.6	3.2
32	0.6	1.2	2.4

Table 1. The $\texttt{VF}/(1+\log_2\delta)$ factor for estimated improvement in number of instructions

We report the speedup factors achieved by an automatically vectorized version over the sequential version of each benchmark, compiled with the same optimization flags. Time is measured using the `getrusage` routine, and includes any overheads introduced by vectorization.

We begin our evaluation with a qualitative analysis of the expected speedups, accompanied by measurements on a set of synthetic test-cases. We use synthetic test-cases to gain a better understanding of the behavior of the vectorized code and of the overheads involved for a range of interleaving factors, some of which are not exhibited in the real-world kernels presented in the following section.

5.1 Profitability of Vectorization in the Presence of Interleaved Data with No Gaps

When data is accessed without gaps, the scalar memory references fully cover the memory range accessed in each iteration. In this case, there is a group of δ scalar-loads accessing δ data-elements interleaved by a factor δ. Our vectorization scheme transforms these loads into a group of δ vector-loads, accessing δ `VF` data-elements, followed by a tree of $\delta\log_2\delta$ `extract` operations. There are therefore $\delta(1+\log_2\delta)$ vector operations compared to δ`VF` scalar operations (that correspond to `VF` scalar loop iterations), yielding a factor of $\texttt{VF}/(1+\log_2\delta)$. Similarly, a group of δ scalar-stores writing δ adjacent data-elements, which are interleaved by a factor δ, is transformed into a tree of $\delta\log_2\delta$ `interleave` operations followed by δ vector-stores writing δ `VF` data-elements, thereby yielding the same factor.

Table 1 presents the values of $\texttt{VF}/(1+\log_2\delta)$ for $\texttt{VF}=4,8,16$ and $\delta=1,2,4,8,16,32$. According to it, vectorization is profitable for all but the two extreme combinations of small $\texttt{VF}=4$ and large stride $\delta=16,32$.

The $\texttt{VF}/(1+\log_2\delta)$ factor takes the total number of instructions into consideration, assuming that each `extract` and `interleave` operation is mapped to a single instruction (as is the case in Altivec). It ignores the fact that `extract/interleave` operations typically execute on a different unit than loads and stores, and the fact that they introduce data dependencies. Overall, this factor could serve as a rough estimate of the speedups we can expect from vectorizing interleaved accesses (both loads and/or stores).

We measured the actual speedups that auto-vectorization obtains for different values of δ and `VF` using two sets of synthetic test-cases that operate on int, short, and char data elements (corresponding to `VF`=4,8,16, respectively). The first set of tests contains only memory operations; data is loaded and immediately stored back in a different order. In the second set, each test contains $\delta-1$ addition operations and only a single store in the loop (storing the sum of the loaded δ elements) to consecutive addresses. Figure 8 shows speedup factors we obtain for these tests. For each δ value, we show the measured results when the data is aligned and unaligned, and the expected improvement factor on the instruction count (the value of $\texttt{VF}/(1+\log_2\delta)$).

According to the qualitative estimation in Figure 8a, vectorization is expected to be profitable for all but the two extreme combinations of small $\texttt{VF}=4$ and large stride $\delta=16,32$. The actual measured speedups we get are somewhat lower in some cases, but the overall behavior is largely similar. According to Figure 8a, vectorization is profitable for $\delta=2$, and for larger values of δ if `VF` is sufficiently large. For the highest interleaving factor of $\delta=32$, performance degrades for all values of `VF` but for $\texttt{VF}=16$. In general, the speedups for $\delta=32$ are smaller compared to the expected values due to register spills that occur in the vectorized version at such a high interleaving factor. The extraction code alone needs δ registers and Altivec has 32 architected vector registers, so register spilling starts to occur at $\delta\geq 32$. The speedups for $\delta=8,16$ are also somewhat lower than expected for some `VF`s. The main reason for this seems to be the large size of the vectorized loop that inhibits loop-unrolling as compared to the scalar version that is being unrolled.

The speedups in Figure 8b are better than the speedups we see in Figure 8a because of the improved Instruction Level Parallelism (ILP). The arithmetic operations hide some of the overhead of the `extract/interleave` operations. Such parallelism can be exploited by the super-scalar/out-of-order executing PowerPC970, which can execute the `extract/interleave` operations on its vector-permute unit while performing the additions on the vector-arithmetic unit.

Overall, these results show that our vectorization scheme is profitable even for interleaving factors as high as 16 and 32, provided there is enough computation relative to the amount of memory access, at least for $\texttt{VF}>4$.

5.2 Profitability When Data is Accessed with Gaps

When data is accessed with gaps, there is a group of $n<\delta$ scalar-loads, each with δ-strided access. Our vectorization scheme still uses δ vector-loads to load δ `VF` data-elements, but not all elements loaded will be used. Vectorization therefore loses some of its effectiveness. In addition, some of the $\delta\log_2\delta$ `extract` operations may not be needed depending on the specific indices of the existing accesses; such useless `extract` operations are discarded by dead-code elimination.

For example, when accessing 2 out of $\delta=8$ elements at indices (0,4) a total of 8 `extracts` are required (e.g., accessing in each iteration elements $a[8{*}i]$ and $a[8{*}i{+}4]$). However, when accessing elements at indices (0,1) a total of 14 `extracts` are required (e.g., $a[8*i]$ and $a[8*i+1]$).

The maximum number of `extracts` required for a group of $n\leq\delta$ loads with stride δ, denoted by $f(n,\delta)$, can be computed recursively. For $n=1$ a single resultant `extract` is needed, which implies that a total of $f(1,\delta)=\sum_{i=0}^{(\log_2(\delta-1))}2^i=\delta-1$ `extracts` are required. For $n=2$, a maximum of $f(2,\delta)=2(\delta-1)$ `extracts` are needed. This occurs when each of the two resultant `extracts` requires a separate set of $\delta-1$ `extracts`. For $n\geq 2$, the maximum number of `extracts` is needed when (roughly) half of the loads access even elements and the rest access odd elements. We need $\delta/2$ `extracts` to separate the even elements and likewise for the odd elements, effectively reducing the desired stride by half. Thus, for $n\geq 2$,

$$f(n,\delta)=\delta+f(\lceil n/2\rceil,\delta/2)+f(\lfloor n/2\rfloor,\delta/2).$$

When $n=\delta$ we need the full set of $f(\delta,\delta)=\delta\log_2\delta$ `extracts`. Table 2 contains the values of $f(n,\delta)$ computed by the above recursive formula for $\delta=2,4,8$ and $n=1,2,\ldots,\delta$.

The number of vector operations for n accesses of stride δ is δ load/stores plus at-most $f(n,\delta)$ `extracts`/`interleaves`, compared to n`VF` scalar load/stores. So the expected speedup factor is at-least $n\texttt{VF}/(\delta+f(n,\delta))$.

We measured the actual speedups obtained on a set of synthetic tests that contain interleaved-loads with $\delta=8$ and arithmetic

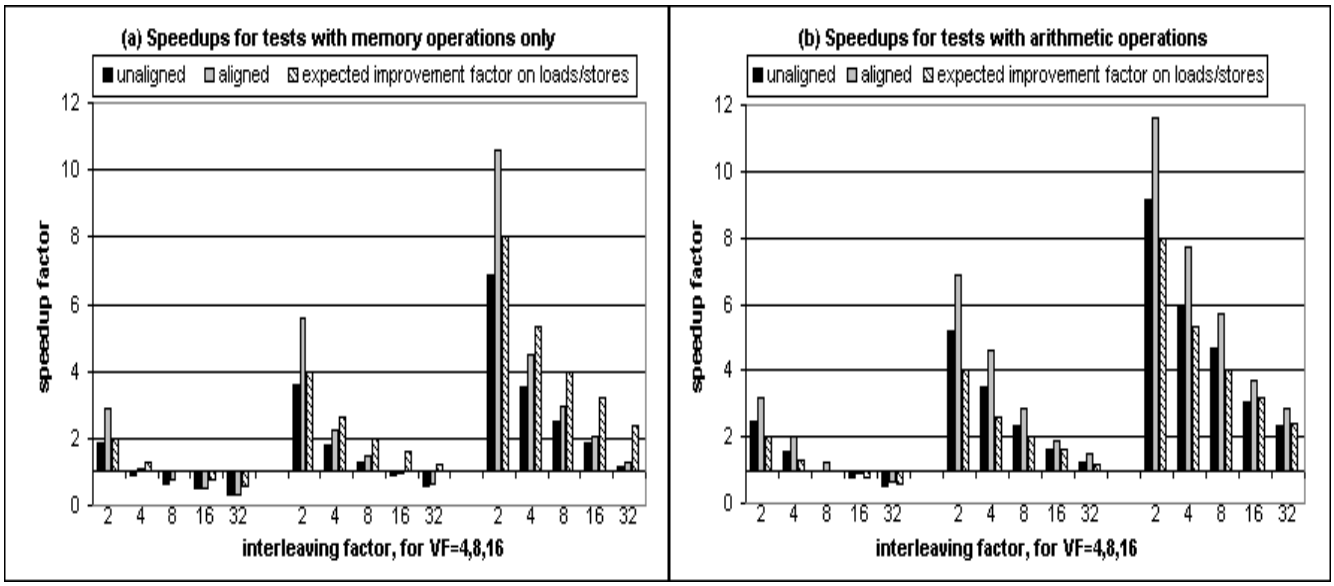

Figure 8. Autovectorization improvements of synthetic kernels with fully-interleaved data

n	$\delta = 2$	$\delta = 4$	$\delta = 8$
1	1	3	7
2	2	6	14
3		7	17
4		8	20
5			21
6			22
7			23
8			24

Table 2. Maximum number of extract operations $f(n, \delta)$ for different values of δ and n

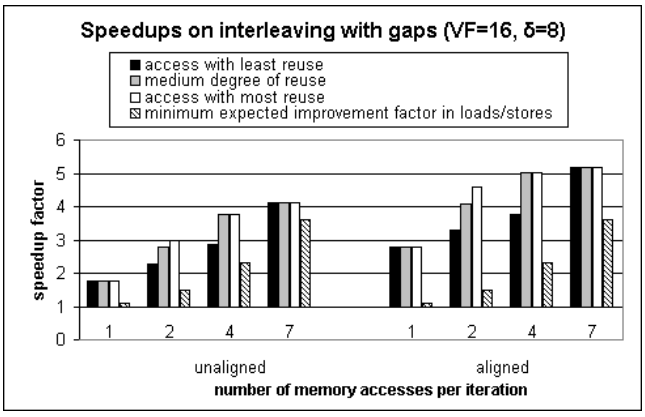

Figure 9. Speedups on synthetic kernels with gaps

operations, for VF $= 16$. The measured speedup factors are shown in Figure 9, along with the worst-case expected factors of reduction in instruction-count (the values for $n\text{VF}/(\delta + f(n, \delta))$ for VF=16). These estimates show a positive improvement even for the extreme $n = 1$ cases.

We experimented with $n = 1, 2, 4, 7$ stride-8 accesses with three different sets of indices. The first set requires the maximum number of `extracts`: for $n = 2$ and $n = 4$, this set has accesses to indices (0,1) and (0,1,2,3), respectively. The second set has an intermediate level of reuse (e.g., indices (0,2) for $n = 2$), and the third set requires the minimum number of `extracts`, having accesses to indices (0,4) and (0,2,4,6) for $n = 2, 4$, respectively. (For $n = 1$, we used an access to index (0) in all three sets, and for $n = 7$ we used accesses to indices (0,1,2,3,4,5,6)).

As can be seen in Figure 9, the measured speedups for the worst-case scenarios are within the expected range, according to the qualitative analysis. The important thing to note about these results is that even when the access to data is relatively sparse and the data reuse is low, vectorization still improves performance.

6. Experimental Results on Real-world Kernels

The next step in our experimental evaluation was to test our vectorization scheme on real-world computations. The test cases we constructed are representative of main computation kernels in real-world applications from different domains. Table 3 briefly describes the kernels used in our experiments. The kernels are named '$i\delta\_name - ts$' indicating that the interleaving factor present in a kernel is δ, and the data type operated upon is signed (if $t = s$) or unsigned (if $t = u$), and is of size s bits. (In some cases, there are two different interleaving factors for the reads and writes in the same kernel.) All kernels except for the last five access fully interleaved data. The five kernels with gaps have the number of accessed elements n as part of their name: '$i\delta\_n\_name - s$'. Generally, kernels operating on unsigned chars (u8) were taken from the video domain (operations on complex data (in which $\delta = 2$) or RGBA images (where $\delta = 4$)). Kernels operating on signed shorts were taken mostly from the audio and communication domain (operating on complex data, or audio streams with 1, 2, or 4 interleaved channels). These tests often involve type-conversions and/or widening integer multiplications that require two vector multiplies per each scalar multiply. Some of the kernels contain reduction operations (summation or maximum) and may not have store operations inside the loop.

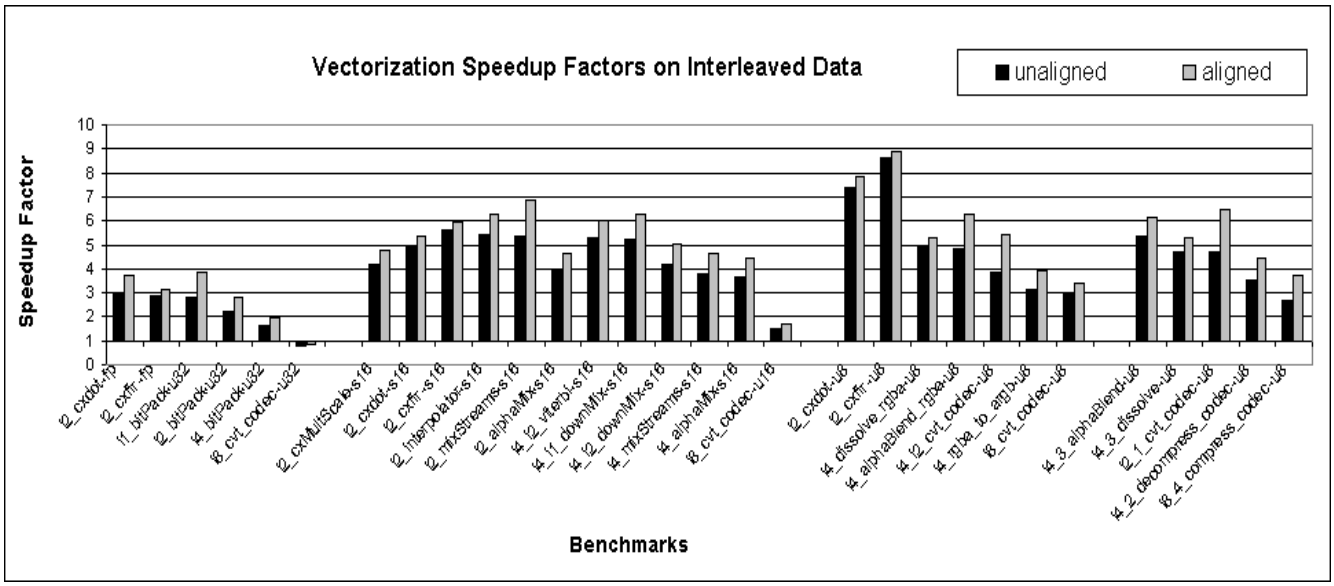

Figure 10. Autovectorization improvements of real-world kernels

Name	Description
i2_cxdot-fp	complex dot product
i2_cxdot-u8	dot product of two video streams
i2_cxdot-s16	dot product of two audio streams
i2_cxfir-fp	complex FIR filter
i2_cxfir-u8	FIR filter on a video stream
i2_cxfir-s16	FIR filter on an audio stream
i1_bitPack-u32	pack 6-bit soft-bits, one per 32bit word
i2_bitPack-u32	pack 6-bit soft-bits, two per 32bit word
i4_bitPack-u32	pack 6-bit soft-bits, four per 32bit word
i2_interpolator-s16	interpolation with up-sampling rate 1:2
i2_mixStreams-s16	mix two stereo streams
i4_mixStreams-s16	mix two 4-channel audio streams
i2_alphaMix-s16	mix two stereo streams with alpha scaling
i4_alphaMix-s16	mix two 4-channel audio streams with alpha scaling
i4_i2_downMix-s16	down mix a 4-channel audio stream to stereo
i4_i1_downMix-s16	down mix a 4-channel audio stream to one channel
i4_i2_viterbi-s16	viterbi decoder
i4_alphBlend-u8	alpha-blend two rgba images
i4_dissolve-u8	image fade away (represented as interleaved rgba)
i2_cxmultScale-s16	complex vector multiply and scale by max value
i4_rgba_to_argb-u8	convert an rgba codec to an argb codec
i4_i2_cvt_codec-u8	decompress a gray codec into an rgba codec
i8_cvt_codec-u32	convert an rrggbbaa codec to an aarrggbb codec
i8_cvt_codec-u16	convert an rrggbbaa codec to an aarrggbb codec
i8_cvt_codec-u8	convert an rrggbbaa codec to an aarrggbb codec
i4_3_alphaBlend-u8	alpha-blend the rgb elements of two rgba images
i4_3_dissolve-u8	fade-away the rgb elements of an rgba image
i2_1_cvt_codec-u8	invert a gray codec
i4_2_decompress_codec-u8	convert a gray codec into an rgba codec
i8_4_compress_codec-u8	convert an rrggbbaa codec to an argb codec

Table 3. Benchmark description

Figure 10 summarizes the speedup factors obtained by applying vectorization to each kernel in two cases. In one case the alignment of the data is unknown and in the other the data is known to be aligned. The results in the figure are organized in four groups; the first three are for fully interleaved data and the last group is of kernels that have gaps.

The first group of six test-cases shows speedups of the floating-point and 32-bit integer kernels, for which $\texttt{VF} = 4$. On our target, we would expect an improvement factor of 2 on the computations (a single 4-way SIMD unit vs. two scalar units). The improvement factor on the loads and stores depends on the interleaving factor δ. For $\delta = 2$, the expected improvement is between 2 (according to the qualitative estimation factor) and 4, assuming some of the `extracts/interleaves` take place in parallel with the computation. Similarly, for $\delta = 4$, the expected improvement on loads/stores is between 1.3 and 4, and for $\delta = 8$, it is between 1 (no improvement) and 4. The speedup factors obtained are within these ranges, achieving 3.7/3.0 on i2_cxdot_fp (aligned/unaligned respectively), 3.1/2.9 on i2_cxfir_fp, and 2.8/2.2 on i2_bitPack_u32. The speedups on the latter are smaller because it is more memory-intensive than the first two. For bitPack, we also show results when there is no interleaving (i1_bitPack_u32), and when $\delta = 4$ (i4_bitPack_u32). It is easy to see how the speedups drop as the interleaving level increases. Last in this group is the kernel i8_cvt_codec-u32, which contains only memory operations and, as expected, degrades performance when vectorized.

The second group of 12 test-cases shows speedups on kernels that operate on short data types, for which $\texttt{VF} = 8$. On our target, we would expect an improvement factor of 4 on the computations (an 8-way SIMD unit vs. two scalar units), and on the loads/stores we would expect a speedup between 4/2.6/2 (according to the qualitative estimation factor) and 8, for $\delta = 2/4/8$ respectively. The speedup factors obtained are within these ranges, achieving an average improvement of 5.6/4.6 (aligned/unaligned respectively) on the $\delta = 2$ kernels and an average improvement of 4.6/3.7 on the $\delta = 4$ kernels. In the kernels i4_i1_downMix-s16 and i4_i2_downMix-s16, we have $\delta = 4$ for the loads (reading a 4-channel stream) and $\delta = 1, 2$ for the stores (writing a single/two channel stream), respectively. The improvements are therefore higher than for the $\delta = 4$ kernels: we get speedups of 6.3/5.3 and 5.0/4.2 (aligned/unaligned) respectively for these two tests. Lastly in this group, we get an improvement factor 1.7/1.5 on our $\delta = 8$ kernel.

A special note should be made about the Viterbi test case, for which we report the overall impact on the total running time of the application, rather than an isolated kernel. The Viterbi decoder computes the most probable transmitted sequence of a convolutional coded sequence. It is used in digital communication data transmission, such as 3G cellular networks and GSM networks. The most computationally intensive part of Viterbi per-

forms a maximization of a likelihood function through a sequence of add-compare-select (ACS) operations, and can benefit significantly from SIMD execution. In each iteration, two input elements are read from two separate arrays, and two consecutive outputs are written into one array. In addition, the data operated upon is interleaved, which altogether amounts to an interleaving factor of 4 upon data write, and an interleaving factor of 2 for each data read. Applying our auto-vectorization to Viterbi, we get a speedup factor of 6.0/5.3. Part of this improvement can be attributed to the usage of vector selects, which are available only on the vector unit for PowerPC970. This helps overcome some of the interleaving handling overhead and gain a significant overall improvement on the entire benchmark due to vectorization.

The third group of seven test-cases shows speedups on kernels that operate on char data types, for which $\texttt{VF} = 16$. On our target, we would expect an improvement factor of 8 on the computations (a 16-way SIMD unit vs. two scalar units). For the loads/stores, the expected speedup is between 8/5.3/4 (according to the qualitative estimation factor) and 16 (assuming some overlap between the `extracts/interleaves` and the arithmetic vector operations) for $\delta = 2/4/8$, respectively. Our experiments show an average improvement of 8.4/8.0 (aligned/unaligned respectively) on the $\delta = 2$ kernels, an average improvement of 5.2/4.2 on the $\delta = 4$ kernels, and an improvement of 3.2/2.9 on the $\delta = 8$ kernel.

The fourth and last group of five test-cases shows speedups on kernels that operate on data with gaps. These kernels operate on char data types (i.e., $VF = 16$) and are expected to get at least the minimum expected improvement factor shown in Figure 9. The qualitative speedups presented there assume the maximum number of `extracts`. The access patterns in these real-world examples actually enjoy the maximal amount of reuse: kernels i4_3_alphaBlend-u8 and i4_3_dissolve-u8 access three out of $\delta = 4$ elements (we get a 6.2/5.4 speedup on alphaBlend (aligned/unaligned) and a 5.3/4.7 speedup on dissolve); i2_1_cvt_codec-u8 accesses one element out of $\delta = 4$ elements in each iteration and is improved by a factor 6.5/4.7; i4_2_decompress_codec-u8 accesses two elements out of $\delta = 4$ elements in each iteration, at indices (0,2); this access pattern enjoys a high-level of reuse and is improved by a factor 4.4/3.5 (aligned/unaligned); and lastly, for the kernel i8_4_compress_codec-u8, we get a speedup of 3.7/2.7 (aligned/unaligned respectively).

7. Discussion

In this section we discuss several opportunities for optimization of special cases, some of which are inspired by comparison to related work.

7.1 Special Cases

There are certain situations that offer an alternative solution to the interleaving problem, deviating slightly from the pure SIMD approach. One example involves computations, such as the down-mixing of a multi-channel audio stream, where each of the interleaved channels is multiplied by a different constant scaling factor and then added together. In such cases, instead of extracting the data elements of each channel into a separate vector, one can prepare in advance a vector holding the different scaling factors and perform the computations on the interleaved data directly, extracting the resultant products as needed.

There are other situations in this spirit that offer similar opportunities to avoid the explicit data reordering, which could be beneficial for the vectorizer to identify. Another example is the following computation:

a[0]+b[0], a[1]-b[1], a[2]+b[2], a[3]-b[3]

(assuming `VF` = 4). To vectorize it, one could first negate `b[1]` and `b[3]` by a proper mask that keeps `b[0]` and `b[2]` intact and then perform one vector add, instead of performing vector add and vector subtract separately on interleaved elements.

Some architectures provide instructions of this SIMOMD (Single Instruction Multiple Operations Multiple Data) flavor in hardware [3, 34]. Loop-based vectorization, which is the approach we follow, targets classic SIMD instructions by packing together identical replicates of the same instruction. In contrast, to generate SIMOMD instructions, an 'SLP' (Superword Level Parallelism) type of approach [17] is needed for grouping different instructions together. The SLP approach provides other interesting alternatives to handling the interleaving problem, as discussed in more detail below.

7.2 Interleaving and the SLP Approach to Vectorization

The SLP approach to vectorization looks for vectorization opportunities in straight-line code. Since its introduction, it has been incorporated into several vectorizing compilers and has provided an interesting aspect for discussion in the context of vectorization.

The SLP approach can handle only limited situations of interleaving in which the interleaved data elements are all manipulated by isomorphic operations, and are accessed without gaps[1]. Despite these limitations, SLP is relevant in the context of vectorization of interleaved accesses because of the special cases of interleaving that it can vectorize and because of the different approach it takes to vectorizing them. Detecting these special cases by our loop-based vectorizer can introduce a new opportunity for their optimization. Furthermore, in some cases, parallelism exists only inside the iteration and not across loop iterations, where SLP is useful but loop-based parallelism is not (e.g., due to loop-carried dependencies). In this subsection, we show how our vectorization approach to interleaving provides the first step towards extending a loop-based vectorizer to perform SLP.

In our 'one-to-one' loop-based approach to vectorization, each vector instruction replaces `VF` instances of a single scalar instruction, corresponding to `VF` consecutive iterations of the loop. In other words, each operation in the loop is considered independently of other operations in the same iteration, but together with its instances across different iterations. In contrast, SLP is a '`VF`-to-one' approach where each vector instruction replaces `VF` distinct scalar instructions, irrespective of any enclosing loop. In order to support interleaved accesses, we had to extend our loop-based vectorizer in the spirit of SLP, in the sense that the vectorizer now considers groups of operations from the same iteration. The analysis was extended to look inside the loop, rather than across different iterations, and to vectorize groups of operations together. Therefore, a loop-based vectorizer that recognizes interleaved accesses provides the first step towards providing SLP capabilities for loops.

In addition to analyzing groups of instructions similar to SLP, our approach also relates to SLP in the way we transform interleaved accesses. The SLP approach starts by looking for groups of accesses to adjacent memory addresses, attempting to pack them together into vector load operations. If such accesses are found inside a countable loop, they are typically strided accesses; however, the stride is across iterations, so SLP does not notice it. In contrast, our approach is to look for groups of δ-strided accesses to adjacent memory addresses in a loop and try to replace them by δ vector load operations. If $\delta = \texttt{VF}$, each vector load we generate essentially replaces `VF` distinct scalar accesses, as does SLP. The same

[1] It is possible to run the first SLP step of searching for a seed of adjacent references repeatedly, each time with a different value for the stride that is considered adjacent; this exhaustive search can be too costly with respect to compile time.

```
for(int i = 0; i < len; i++){
    c[i] = a[2i]+b[2i];
    d[i] = a[2i+1]+b[2i+1];
}
```

Figure 11. Postponing interleaving, scalar

```
for(int i = 0; i < len; i+=VF){
    vector a1 = load(a[2i],a[2i+1],...,a[2i+VF-1]);
    vector a2 = load(a[2i+VF],a[2i+VF+1],...,a[2i+2VF-1]);
    vector b1 = load(b[2i],b[2i+1],...,b[2i+VF-1]);
    vector b2 = load(b[2i+VF],b[2i+VF+1],...,b[2i+2VF-1]);
    vector ab1 = a1 + b1;
    vector ab2 = a2 + b2;
    vector abo = extract_odds(ab1,ab2);
    vector abe = extract_evens(ab1,ab2);
    c[i,i+1,...,i+VF-1] = abo;
    d[i,i+1,...,i+VF-1] = abe;
}
```

Figure 12. Postponing interleaving, vectorized

holds for store operations. This is another reason why our loop-based approach can be viewed as a first step towards applying SLP to loops.

Lastly, the interleaving support we added allows our loop-based vectorizer to handle computations that until now required SLP. This is because the domain of computations that our vectorization scheme can now target also includes unrolled loops, which have traditionally been strictly in the SLP domain.

While we have made the first step towards supporting SLP in loops, we are still essentially using the cross-iteration approach when transforming the loop. By continuing in this direction, we hope to enhance the loop-based vectorizer to provide more efficient support for some special cases of interleaving, described below.

7.3 Towards Loop-aware Interleaving-based SLP

After identifying opportunities to replace a group of scalar loads by a vector load, SLP examines the corresponding dependent instructions in an attempt to replace them with a vector instruction. This usually requires that all scalar instructions be isomorphic (e.g., all are additions), mutually independent, and can all be moved to one place. A loop-based vectorizer could also examine the uses of interleaved loads, and if the above conditions hold, postpone the rearrangement of data to a later stage. For example, the loop in Figure 11 can be vectorized by rearranging the odd and even elements of `a` and `b` separately (as in Figure 3), but it is more efficient to rearrange the odd and even elements of `a+b` instead (see Figure 12). Therefore, a loop-based vectorizer is free to decide when to rearrange the data in such cases. This may be immediately when loading or at a later stage of the computation not originally associated with loads or stores. This resembles the rearranging of data to handle alignment, specifically the *zero shift* versus *lazy shift* heuristics [9]. Rather than immediately reordering the results of loads as in the zero-shift approach for handling alignment, reordering can be postponed and associated with internal operations as in the lazy-shift heuristic. Ultimately, we should reorder in anticipation of the permutation needed by the stores, if internal operations are isomorphic, similar to the *eager shift* scheme. A simple case to optimize occurs when the rearrangement at the stores is the inverse of that at the loads (e.g., `interleave_low/high` and `extract_odd/even`), thereby cancelling each other.

The key point is that performing these optimizations requires extending our interleaving support to look into the uses of the interleaved accesses, and in that sense to continue extending our analysis scheme in the direction of SLP. Considering the uses of interleaved accesses could also provide the opportunity to operate on less than δ`VF` elements in each iteration, thereby reducing the number of `interleaves/extracts` needed. In general, such optimizations may require the use of permute operations that are more flexible than the `interleave_high/low` and `extract_even/odd` that we introduced.

Finally, we wish to emphasize that extending a loop-based vectorizer in the direction described above creates a hybrid 'loop-aware-SLP' approach that can exploit parallelism both across loop iterations as well as inside the loop. This is done by carefully unrolling loops and generating efficient vector instructions and reordering operations. The interleaving support we presented in this paper for a classic loop-based vectorizer provides the first step to realizing this.

8. Related Work

Several works in the field of automatic vectorization have addressed at least some aspect of data reordering.

Vectorizing computations that access non-unit stride data motivated the development of the SIMdD (Single Instructions on Multiple disjoint Data) model and SIMdD architectures, such as the eLite DSP of IBM [20]. Such architectures have better support for reordering vector data than traditional SIMD and can benefit from advanced compiler technology [21]. Using the special vector-pointers of eLite, the vectorizing compiler presented in this work was able to address a wide range of non-unit-stride accesses. Here we show that data reordering for non-unit stride accesses can also be efficient on standard SIMD platforms, using standard compiler infrastructure.

Some SIMD architectures provide capabilities to pack and unpack data, to and from vector registers, during memory operations [2, 6, 10, 35]. Reordering within memory operations increases the already long latency of memory operations. This may cause a series of cache misses when addressing several remote locations and overlooks spatial reuse opportunities [38]. Vectorizing compilers for such architectures are still 'a challenge' [35, 10]. Our focus was to analyze and exploit spatial reuse explicitly by the compiler, for relevant platforms that exist today.

In Section 7, we referred to special cases of interleaving that can be vectorized using SIMOMD operations, such as those targeted by [3, 18]. Also in Section 7, we discussed in detail the interrelations between our work and SLP [17, 33]. These works do not address the general problem of interleaved data, but the way they handle special cases can be incorporated as an extension to our work.

Recent work on classical SIMD [9, 39] focuses on unit-stride accesses, handling the alignment problem that is related to the interleaving problem we address. This includes applying interprocedural analysis to propagate alignment information [29]. The Intel 8.0 compilers [5, 4] can vectorize 2-strided accesses and in particular accesses to complex data (e.g., the example in Figure 1), targeting SIMOMD instructions available in SSE3 [34]. Recent studies focus on vectorizing computations that involve data permutation: Kudriavtsev and Kogge [16] followed the SLP approach for contiguous data, using an abstract source-to-source framework; Ren, Wu and Padua [31] focus on optimizing sequences of permutations. To the best of our knowledge, our effort offers the first general treatment of vectorizing power-of-2 strided accesses in a production compiler that is applicable across classical SIMD platforms.

Vectorization can also be provided by source-to-source optimizers such as VAST [36] which also handles complex data, and SWARP [26] which handles unit stride data. Our aim is to integrate the handling of interleaved data in a standard compiler framework, while estimating performance impacts of additional extract/interleave instructions, their parallelism and register consumption. We aim to show that vectorizing for strides larger than 2 can be beneficial for standard SIMD platforms contrary to a general belief [37].

9. Conclusions

We extended a classic loop-based vectorizer to handle computations with non-unit stride accesses to data, where the strides are powers of 2. Counter to general belief, strided accesses can be vectorized efficiently using standard loop-based SIMD for strides as large as 16 and 32, provided the vectorization factor is sufficiently large and that additional operations exist to hide some of the overhead. This holds even if the data is accessed with gaps, whose impact on performance can be estimated and considered during vectoriztion. Our experiments show that vectorization improves performance of real-world kernels with stride as high as $\delta = 8$, with speedups of 3.2 and 1.7 for `VF`=16 and 8, respectively.

Our implementation uses generic operations for reordering data that are supported on many SIMD platforms today and can handle power-of-2 strides efficiently. We have incorporated our implementation into the open-source GCC compiler, which supports a vast range of platforms, making it freely available for future research and development. The approach of extending loop-based vectorization to handle interleaved accesses can serve as a first step towards a hybrid *loop-aware SLP*, which can exploit parallelism across loop iterations as well as inside loops more efficiently than standard loop-based and SLP vectorization techniques separately.

Acknowledgments

We thank the anonymous referees for their constructive comments and assistance in improving the presentation.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures - A Dependence-based Approach*. Morgan Kaufmann Publishers, 2001.

[2] K. Asanovic and D. Johnson. Torrent Architecture Manual. Technical report tr-96-056, Internation Computer Science Institute (ICSI), 1996.

[3] L. Bachega, S. Chatterjee, K. A. Dockserz, J. A. Gunnels, M. Gupta, F. G. Gustavson, C. A. Lapkowskix, G. K. Liu, M. P. Mendell, C. D. Wait, and T. J. C. Ward. A High-performance SIMD Floating Point Unit for BlueGene/L: Architecture, Compilation, and Algorithm Design. In *Proc. of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT'04)*, pages 85–96, September 2004.

[4] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian. Efficient exploitation of parallelism on Pentium III and Pentium 4 processor-based systems. *Intel Technology J.*, February 2001.

[5] A. Bik. *The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance*. Intel Press, 2004.

[6] J. Corbal, R. Espasa, and M. Valero. Exploiting a New Level of DLP in Multimedia Applications. In *Proc. of the 32nd annual ACM/IEEE International Symposium on Microarchitecture (Micro)*, pages 72–79, 1999.

[7] P. D'Arcy and S. Beach. StarCore SC140: A New DSP Architecture for Portable Devices. In *Wireless Symposium*. Motorola, September 1999.

[8] K. Diefendorff, P. K. Dubey, R. Hochsprung and H. Scales. Altivec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, Vol. 20, No. 2, pages 85–95, March-April 2000.

[9] A. E. Eichenberger, P. Wu, and K. O'brien. Vectorization for SIMD Architectures with Alignment Constraints. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 82–93, June 2004.

[10] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: A Vector Extension to the Alpha Architecture. In *Proc. of the 29th Annual International Symposium on Computer Architecture (ISCA)*, pages 281–292, May 2002.

[11] Free Software Foundation. Auto-Vectorization in GCC, http://gcc.-gnu.org/projects/tree-ssa/vectorization.html.

[12] Free Software Foundation. GCC, http://gcc.gnu.org.

[13] G. Goff, K. Kennedy, and C. Tseng. Practical Dependence Testing. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 15–29, June 1991.

[14] Texas Instruments. www.ti.com/sc/c6x, 2000.

[15] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49(4), pages 589–604, July 2005.

[16] A. Kudriavtsev and P. Kogge Generation of Permutations for SIMD Processors *in Proc. of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES)*, pages 147 – 156, June 2005.

[17] S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 145–156, June 2000.

[18] J. Lorenz, S. Kral, F. Franchetti, and C. W. Ueberhuber. Vectorization Techniques for the BlueGene/L Double FPU. *IBM Journal of Research and Development*, 49(2-3), pages 437–446, March/May 2005.

[19] J. Merrill. Generic and Gimple: A New Tree Representation for Entire Functions. In *the GCC Developer's summit*, pages 171–180, June 2003.

[20] J. H. Moreno, V. Zyuban, U. Shvadron, F. Neeser, J. Derby, M. Ware, K. Kailas, A. Zaks, A. Geva, S. Ben-David, S. Asaad, T. Fox, M. Biberstein, D. Naishlos, and H. Hunter. An Innovative Low-power High-performance Programmable Signal Processor for Digital Communications. *IBM Journal of Research and Development* 47(2-3), pages 299–326, March/May 2003.

[21] D. Naishlos, M. Biberstein, S. Ben-David, and A. Zaks. Vectorizing for a SIMdD DSP Architecture. In *Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 2–11, October 2003.

[22] D. Naishlos and R. Henderson. Multi-platform Auto-vectorization. In *Proc. of the 4th Annual International Symposium on Code Generation and Optimization (CGO)*, March 2006.

[23] H. Nguyen and L. K. John. Exploiting SIMD Parallelism in DSP and Multimedia Algorithms using the AltiVec Technology. In *Intl. Conf. on Supercomputing*, pages 11–20, 1999.

[24] D. Novillo. Tree SSA - a New Optimization Infrastructure for GCC. In *Proc. of the GCC Developers Summit*, pages 181–194, June 2003.

[25] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro* Vol.16, No.4, pages 42–50, August 1996.

[26] G. Pokam, S. Bihan, J. Simonnet, and F. Bodin. SWARP: A Retargetable Preprocessor for Multimedia Instructions In *Concurrency and Computation: Practice and Experience; Special Issue: Compilers for Parallel Computers*, Vol. 16, No. 2-3, pages 303 – 318, January 2004.

[27] S. Pop, G. Silber, A. Cohen, P. Clauss, and V. Loechner. Fast Recognition of Scalar Evolutions on Three-address SSA Code. Research Report A/354/CRI, CRI/ENSMP, April 2004.

[28] S. Pop, A. Cohen, and G. Silber. Induction Variable Analysis with Delayed Abstractions. In *Proc. of the First International Conference of High Performance Embedded Architectures and Compilers (HiPEAC)*, pages 218–232, November 2005.

[29] I. Pryanishnikov, A. Krall, and N. Horspool. Pointer Alignment Analysis for Processors with SIMD Instructions. In *Proc. of the 5th Workshop on Media and Streaming Processors at Micro '03*, pages 50–57, December 2003.

[30] G. Ren, P. Wu, and D. Padua. A Preliminary Study on the Vectorization of Multimedia Applications for Multimedia Extensions. In *16th International Workshop of Languages and Compilers for Parallel Computing (LCPC)*, pages 420 – 435, October 2003.

[31] G. Ren, P. Wu, and D. Padua. Optimizing Data Permutations for SIMD Devices. to appear in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2006.

[32] J. Shin, J. Chame, and M. W. Hall. Compiler-controlled Caching in Superword Register Files for Multimedia Extension Architectures. In *Proc. of the 11th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 45–55, September 2002.

[33] J. Shin, M. Hall, and J. Chame. Superword-Level Parallelism in the Presence of Control Flow. In *Proc. of International Symposium on Code Generation and Optimization (CGO)*, pages 165–175, March 2005.

[34] K. B. Smith, A. J. Bik, and X. Tian. Support for the Intel Pentium 4 Processor with Hyper-threading Technology in Intel 8.0 Compilers. *Intel Technology Journal*, 8(1), pages 19–31, February 2004.

[35] D. Talla, L. K. John, and D. Burger. Bottlenecks in Multimedia Processing with SIMD Style Extensions and Architectural Enhancements. *IEEE Trans. on Computers* Vol. 52, No. 8, pages 1015–1031, August 2003.

[36] Crecent Bay Software. VAST-F/ALtivec: Automatic Fortran Vectorizer for PowerPC Vector Unit, http://www.crescentbaysoftware.com/-docs/vastfav.pdf.

[37] Crecent Bay Software. Vast/altivec faq: Vectorization for Altivec, http://www.crescentbaysoftware.com/altivec_FAQ.html.

[38] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, 1996.

[39] P. Wu, A. E. Eichenberger, and A. Wang. Efficient SIMD Code Generation for Runtime Alignment. In *Proc. of the International Symposium on Code Generation and Optimization (CGO)*, pages 153 – 164, March 2005.