H-0235 (H0512-001) November 30, 2005
Computer Science

IBM Research Report

Auto-Vectorization of Interleaved Data for SIMD

Dorit Nuzman, Ira Rosen, Ayal Zaks
IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

Auto-Vectorization of Interleaved Data for SIMD

Dorit Nuzman   Ira Rosen   Ayal Zaks

IBM Haifa Labs, Haifa, 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. In this paper we demonstrate an automatic compilation scheme that supports effective vectorization in the presence of interleaved data with strides that are power of 2, facilitating data reorganization. We demonstrate how our vectorization scheme applies to SIMD architectures that are dominant today, and present experimental results on a wide range of key kernels, showing speedups up to 3.7 for interleaving level (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, 13], gaming machines [14], general purpose processors [23, 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, scientific codes and more, by simultaneously executing the same instruction on multiple data elements.

Optimizing compilers use vectorization techniques [31] 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 [26] 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, with additional alignment restrictions. Computations, on the other hand, may access data elements in an order which 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. In addition, these mechanisms widely differ 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 strides that are power of 2, while exploiting spatial reuse.
- Demonstrate the applicability of our technique 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 shows 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 that it raises, and the abstractions we introduced to express the unique data reordering features of SIMD platforms. In Section 3 we give a brief overview of the GCC compiler we used with its infrastructure, and present the abstractions we introduced to express the unique data reordering features of SIMD platforms. Section 4 describes the extensions we introduced to the vectorizer to handle interleaved data. Experimental results obtained by running the vectorizer are presented in Section 5. Section 6 brings up an interesting connection between our work and SLP. Section 7 discusses related work and Section 8 concludes.

2. The Interleaving Problem

Vector instructions of conventional SIMD extensions access multiple data that is adequately packed in vector registers. In contrast,

```
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+4,2i+6] = abee - aboo;
    c[2i+1,2i+3,2i+5,2i+7] = abeo + aboe;
}
```

Figure 2. Vectorized complex multiplication

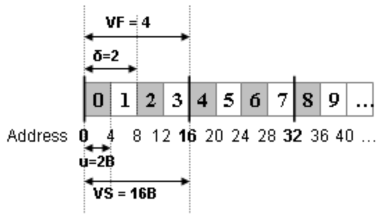

Figure 3. Illustration of the memory access pattern of the code example

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 provide the ability to load and store multiple data from memory only if the addresses a_i are consecutive, i.e. are 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 3).

This implies that in order to access the odd and even elements separately, they need to be extracted after loading them both consecutively into vector registers. A similar case applies to storing elements to non consecutive addresses (see figure 4).

The above example 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 illustration of the functionality of these idioms in Figure 5). Such access patterns are generally called *non-unit stride* accesses, of the form $a_i = b + \delta ui$

```
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 4. Vectorized complex multiplication with data reordering

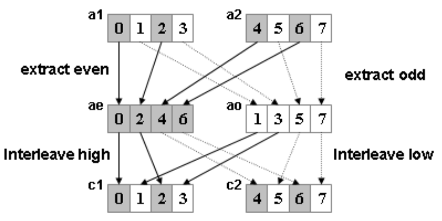

Figure 5. Illustration of the extract_even/odd and interleave_low/high operations

for some constant $\delta > 1$ called the *interleaving factor*. As we shall see later, these extract and interleave abstractions play an important role in handling strided accesses by the compiler.

Notice that each scalar iteration can have n different memory accesses into the range of δ elements. When $n = \delta$ we say that the access pattern is *fully interleaved*, and when $n < \delta$ we say that the interleaved access-pattern contains *gaps* (in our code example in Figure 1 we have $n = \delta = 2$). 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 to load/store $\delta\texttt{VF}$ consecutive (and aligned) elements, from which the desired VF elements for the original access are extracted/inserted. Notice that several (n) vector accesses can extract their elements from the set of $\delta\texttt{VF}$ consecutive elements; It is important to recognize interleaved (and in particulr fully interleaved) data to exploit spatial locality, and also to facilitate stores which typically cannot tolerate gaps.

In this paper we deal with the interleaving problem, which is to automatically recognize interleaved accesses and generate optimized code to efficiently vectorize them. Notice that the general scheme of loading additional elements and then extracting the desired elements is also used to solve the ubiquitous alignment problem [9, 32]. 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. We restrict our attention to strided accesses because their overhead can be reduced 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 that is being targetted. 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 for a unique architectural support to 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 [23, 28]. In recent years, a variety of implmenetations of the SIMD model have been incorporated into many platforms, most of which offer special mechanisms to shuffle data between vector registers [2, 8, 21, 23, 27]. These mechanisms incure 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 an 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) will extract the elements at these indices from the concateneated vector a1,a2. An important thing to notice about the permute idiom is that on most platforms with SIMD support (MMX, SSE, MIPS, IA-64 and SPE) this operation is not supported in its generic form [20], and usually requires immediate values to the permutation mask, or allow 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 (illustrated in Figure 5) rather than a generic shuffle/permute idiom. (Other reasons for this choice will be revealed in Section 4.2).

3. Vectorizer Overview

The compiler we used for implementing the vectorization of computations that involve interleaved data is GCC (the GNU Compiler Collection [11]) of the Free Software Foundation. In this section we briefly describe the infrastructure of GCC that is relevant to our work.

GCC uses multiple levels of Intermediate Languages (IL) in the course of translating the original source language to assembly. Our focus is on a new IL called GIMPLE [17] which supports Static Single Assignment (SSA) [22] and retains enough information from the source language to facilitate advanced data-dependence analysis and aggressive high-level optimizations including auto-vectorization [10]. From the high-level target-independent GIMPLE IL, statements are translated to the low-level instructions of the RTL IL (Register Transfer Language), where target-dependent optimizations such as instruction scheduling and register allocation are applied.

Figure 6 outlines the vectorization pass of GCC. The vectorizer applies a set of analyses to each loop (see `vect_analyze_loop()`), followed by the actual vector transformation (`vect_transform_loop()`) for loops that successfully completed the analysis phase

The first analysis phase, `analyze_loop_form()` identifies innermost, single basic block countable a loops. (Certain multiple basic block constructs may be collapsed into a 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).

```
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

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.

Next, `analyze_data_refs()` finds all the memory references in the loop and checks if they are "analyzable", meaning that an access function describing the series of addresses across iterations of the loop can be constructed (using scalar evolution analysis [24, 25]). This is needed for memory-dependence, access-pattern and alignment analyses.

Dependence cycles represent recurrences that have to be handled in order to enable vectorization of a loop. The function `analyze_scalar_cycles()` examines dependence cycles which involve scalar variables (i.e. do not go through memory), such as reductions, and verifies that they can be handled appropriately. 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 dependences) or at-least `VF`, by applying a set of classic data dependence tests [1, 12, 31].

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. The `analyze_data_ref_alignment()` phase makes sure that 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. 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 top-down (vectorizing defs before their uses), inserting a vector statement in the loop for each scalar statement that needs to be vectorized.

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 like reductions, alignment-support mechanisms and vector element shuffling operations (such as the `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, new platform-independent vector abstractions had to be introduced 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 which do not divide by the vectorization factor (this epilog also must 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 that we introduced to the `vect_analyze_loops()` and `vect_transform_loops()` phases of the vectorizer in order to extend it to 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

A new data-structure was introduced to represent a group of load (or store) instructions, which we refer to simply as a *group* in what follows. We assign a (possibly negative) integer i called the *index* to each member of a group, such that $i_1 - i_2 = b_1 - b_2$ for any two members of a group. These indices help determine the relative reordering needed by the members of the group. Note that members of a group have distinct indices; there shouldn't be multiple members with the same index if redundant-load-and-store elimination collapsed pairs of loads from the same address into a single load and removed the first of two stores to the same address (prior to vectorization). 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 will later be responsible for loading or storing the data for the group — the base address of the leader determines the overall starting address; all other members will use the difference between their index and the leader's index to determine the reordering 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 enable quick navigation from group members to their leader and to their index. We now describe how we construct the groups efficiently.

A pair of loads or stores can be assigned to the same group if they have the same interleaving factor and start at relatively close base addresses. Specifically, spatial locality occurs for pairs of accesses of the form $a_1(i) = b_1 + \delta_1 u_1 i$ and $a_2(i) = b_2 + \delta_2 u_2 i$, where $\delta_1 u_1 = \delta_2 u_2$ and either $b_1 - b_2 \le \mathtt{VS} - u_1$ or $b_2 - b_1 \le \mathtt{VS} - u_2$. Most occurrences of interleaved accesses involve the same unit sizes $u_1 = u_2$, so we simplify the conditions to

$$\delta_1 = \delta_2 \text{ and} \tag{1}$$

$$|b_1 - b_2| < \mathtt{VS}. \tag{2}$$

A group of loads or stores can be assigned to the same group by analyzing them in pairs and clustering them together. Our approach is to analyze pairs of loads and stores in such an order that keeps one load or store fixed as a pivot, so that a group is augmented by one new member at a time. 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 also detect pairs of loads and stores for grouping:

```
if (distance_between_accesses < VS)
  if (read,write): goto dep_resolve.
  if (read, read): analyze_interleaving; done.
  if (write, write): analyze_interleaving;
            goto dep_resolve.
else
  done
```

We first check if the distance is smaller than `VS`, which is needed for both recognizing problematic dependencies and exploiting spatial reuse (Condition (2)). In the case of a load and a store, there is a dependence to check (via `dep_resolve`) but we do not try to group them together. Pairs of loads do not have a dependence but can be grouped together (via `analyze_interleaving`), and finally both considerations apply to a pair of stores.

The function `analyze_interleaving` examines a pair of non-unit stride accesses that meet Conditions (1) and (2), and builds interleaving groups incrementally as follows. If both accesses have not been assigned to groups yet, we open a new group for them and assigned indices $i_1 = 0$ and $i_2 = i_1 + b_2 - b_1$. We make sure that $|i_2 - i_1| \le \mathtt{VF}$, which is the common and simpler case to handle. We also record the access with the smaller base address as the leader of the group, and record the index of the other access. Otherwise, if one access (`s1`) has been already assigned to a group and the other (`s2`) has not, we assign `s2` to the group of `s1` and set the index of `s2` to be $i_2 = i_1 + b_2 - b_1$. We then check if i_2 is smaller than the index of the leader and if so make `s2` the new leader, and also update the largest index found in the group if i_2 is larger than it. If either cases hold, we check that $i_l - i_s \le \mathtt{VF}$ where i_l is the (new) largest index and i_s is the index of the (new) leader. 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 implies that the complexity of the grouping algorithm is linear in the number of loads and stores.

After all pairs of loads and stores have been traversed we need to visit each group and estimate the profitability of vectorizing it, including the identification and assessment of gaps — they reduce the amount of reuse and may by intolerable for stores (see Section 5.1 for 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 anyhow to permit non-unit stride accesses.) To do so we use the leaders of the groups — every time we reach a leader during the `analyze_data_ref_accesses()` scan, we analyze its group. During this scan we also look for strided accesses that do not belong to any group, and if found we open (and analyze) new singleton groups for them.

4.2 Extensions to transformation phase

We now explain how the transformation phase was modified to vectorize interleaved data, based on the groups produced by the enhanced analysis phase. When reaching the first member of a group we generate δ load or store statements according to the address of the leader. 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 [20]). We then generate a set of data reordering statements of the form `extract_even/odd` (for loads) and `interleave_low/high` (for stores), according to the stride of the group. 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 4 loads by a combination of `extract_evens`, as depicted in Figure 7. The complete set of $\delta \log_2 \delta$ `extract_even` and `extract_odd` statements are generated after the vector loads at the location of

```
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`

the first scalar load. Each member of the group is then connected to the appropriate resultant `extract_odd/even` statement. The `extract_odd/even` statements that remain unused, in the case of gaps, will be discarded by a later dead-code elimination pass.

The case of stores to interleaved data is handled 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 extensions we introduced to the vectorizer to handle interleaved data — no modifications were needed for handling the vectorization of non-load/store statements.

For strides that are not a power of 2, permute capabilities more flexible than `extract_odd/even` and `interleave_low/high` are needed, which are provided by some platforms. In addition, each non-power-of-2 stride generally requires a tailored, irregular solution. The simple and generic `extract_odd/even` and `interleave_low/high` abstractions, on the other hand, 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.

It is worth noting that one could alternatively have the vectorizer disregard spatial reuse, generate the required treatment for each instruction separately and rely on a subsequent redundant loads and stores 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 which 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 possible.

5. Experimental Results

The results we present in this section were generated automatically using the autovect-branch of the GCC 4.1 compiler, freely available from the FSF [11]. Experiments were performed on an IBM PowerPC 970 processor having Altivec support, running under Linux. The PowerPC 970 is an out-of-order execution super-scalar processor with 5 scalar functional units (2 fixed point, 2 floating point, and 1 branch), 2 SIMD units (1 arithmetic, 1 permute) and 2 memory units shared between the scalar and vector units. 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.

5.1 Qualitative analysis

We start our evaluation with a qualitative analysis of the expected speedups, accompanied by measurements on a set of synthetic test-cases. We use it 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 that we present in the following subsection.

5.1.1 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 load-accesses 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 δVF scalar operations (that correspond to VF scalar loop iterations), yielding a factor of $\mathtt{VF}/(1 + \log_2 \delta)$. Similarly, a group of δ scalar-stores writing δ adjacent data-elements that 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.

The $\mathtt{VF}/(1 + \log_2 \delta)$ 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. But overall, this factor could serve as a rough estimate of the speedups to 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. The speedup factors we obtain for these tests are shown in Figure 8. For each δ value we show the measured results when the data is aligned and unaligned, and also the expected improvement factor on the instruction count (the value of $\mathtt{VF}/(1 + \log_2 \delta)$).

As can be seen in Figure 8a, according to the qualitative estimation, vectorization is expected to be profitable for all but the two extreme combinations of small $VF = 4$ 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 the VF is sufficiently large. For the highest interleaving factor of $\delta = 32$ performance degrades for all values of VF. 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 VFs. The main reason for that seems to be the large size of the vectorized loop that inhibits loop-unrolling compared to the scalar version that is being unrolled.

The speedups we see 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 if there

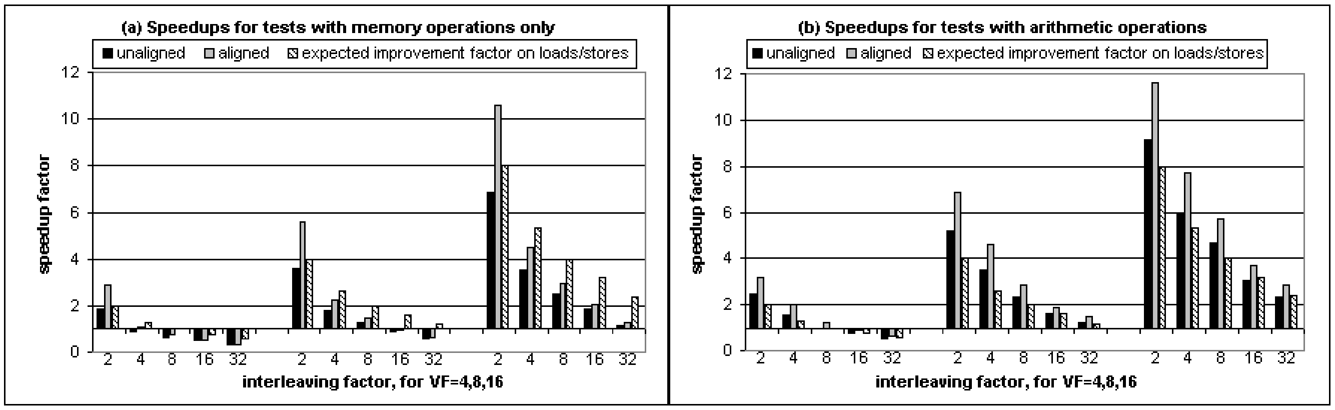

Figure 8. Autovectorization improvements of synthetic kernels with fully-interleaved data

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

is enough computation relative to the amount of memory access, at least for VF > 4.

5.1.2 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 looses some of its effectiveness due to this. In addition, some of the $\delta \log_2 \delta$ `extract` operations may not be needed (those are discarded by dead-code elimination), depending on the specific indices of the existing accesses.

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

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 as follows. For $n = 1$ a total of $f(1, \delta) = \sum_{i=0}^{(\log_2(\delta-1))} 2^i = \delta - 1$ `extracts` is required. For $n = 2$, a maximum of $f(2, \delta) = 2(\delta - 1)$ `extracts` are needed. 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 $f(n, \delta) = \delta + f(\lceil n/2 \rceil, \delta/2) + f(\lfloor n/2 \rfloor, \delta/2)$ for $n \geq 2$. Table 1 contains the values 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 at-most δ loads or stores plus $f(n, \delta)$ `extracts` or `interleaves`, compared to nVF scalar loads or 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 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 $n\texttt{VF}/(\delta + f(n, \delta))$ for VF=16). Notice that these estimates show a positive improvement even for the extreme $n = 1$ cases.

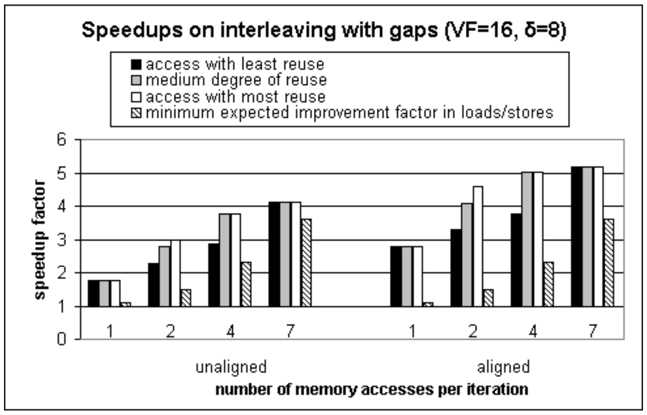

Figure 9. Speedups on synthetic kernels with gaps

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 the Figure, 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.

5.2 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 the main computation kernels in real world applications from different domains. Table 2 briefly describes the kernels used in our experiments. The kernels are named

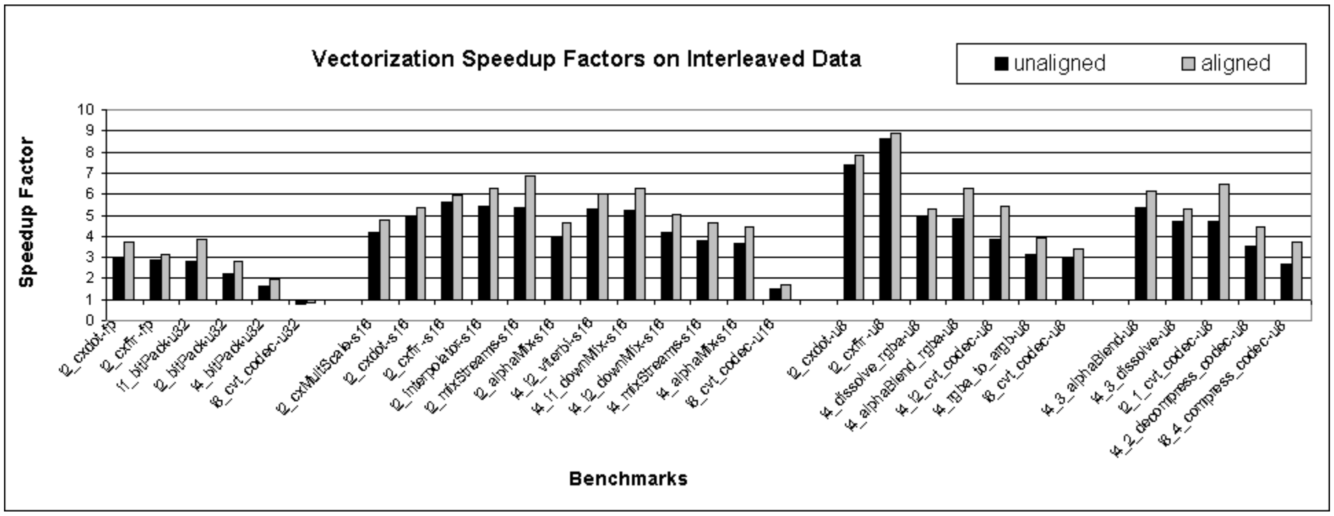

Figure 10. Autovectorization improvements of real-world kernels

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 2. Benchmark description

“$i\delta\_name - ts$” 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 the writes in the same kernel). All kernels except for the last 5 access fully interleaved data. The 5 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 which 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 summarizes the speedup factors obtained by applying vectorization to each kernel in two cases — one in which the alignment of the data is unknown, and the other in which the data is aligned. The results in the figure are organized in 4 groups — the first three are for fully interleaved data, and the last group is of kernels that have gaps.

The first group of 6 test-cases shows speedups of the floating-point and 32 bit integer kernels, for which $VF = 4$. On our target we would expect an improvement factor of 2 on the computations (a single 4-way SIMD unit vs. 2 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 to 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 $VF = 8$. On our target we would expect an improvement factor of 4 on the computations (an 8-way SIMD unit vs. 2 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, acheiving an avereage 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 testcase, for which we report the overall impact on the total running time the application, rather than an isolated kernel. The Viterbi decoder computes the most probable transmitted sequence of a convolutional coded sequence, used in digital communication data transmission, such as 3G cellular networks and GSM networks. The most computationally intensive part of Viterbi performs 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 seperate arrays), and two consecutive outputs are written (into one array). In addition, the data operated upon is interleaved, which all together amounts to interleaving factor 4 upon data write, and interleaving factor 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 has to be attributed to the usage of vector selects, that on PPC970 are available only on the vector unit. This helps to overcome some of the interleaving handling overhead, and gain a significant overall improvement on the entire benchmark due to vectorization.

The third group of 7 test-cases shows speedups on kernels that operate on char data types, for which $VF = 16$. On our target we would expect an improvement factor of 8 on the computations (an 16-way SIMD unit vs. 2 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 8.4/8.0 average improvement (aligned/unaligned respectively) on the $\delta = 2$ kernels, a 5.2/4.2 average improvement on the $\delta = 4$ kernels and a 3.2/2.9 improvement on the $\delta = 8$ kernel.

The fourth and last group of 5 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. Recall that 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 3 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 1 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 2 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); Lastly, for the kernel i8_4_compress_codec-u8 we get a speedup of 3.7/2.7 (aligned/unaligned respectively).

6. Discussion

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

6.1 Special Cases

There are certain situations that offer an alternative solution to the interleaving problem, deviating slightly from the pure SIMD approach. One example are 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 on interleaved elements separately.

Some architectures provide instructions of this SIMOMD (Single Instruction Multiple Operations Multiple Data) flavor in hardware [3, 28]. Notice that 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 [15] 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.

6.2 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 limitation, 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 open a new opportunity to optimize them. 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 (due to loop-carried dependencies for example). 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 it’s 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

[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

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 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 using essentially the cross-iteration approach when transforming the loop. The next observation we want to make is that by continuing in this direction we can enhance the loop-based vectorizer to provide more efficient support for some special cases of interleaving, which we now describe.

6.3 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 too 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 rearranging 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 4), but it is more efficient to rearrange the odd and even elements of `a+b` instead (see Figure 12). So a loop-based vectorizer is free to decide when to rearrange the data in such cases — immediately when loading or at a later stage of the computation (which is not originally associated with loads or stores). This resembles the issue of rearranging data to handle alignment, specifically the *zero shift* versus *lazy shift* heuristics [9]. That is, rather than immediately reordering the results of loads as in the zero-shift approach for handling alignment, reordering can be postponed and be 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`).

The key point to notice here is that performing these optimizations requires extending our interleaving support to look into the 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 δVF elements in each iteration, thereby reducing the number of `interleaves/extracts` needed. The interleaving-support mechanism we envision can be viewed as a *funnel* having the loads on top, going down through `extracts` into a series of isomorphic computations, which are followed by `interleaves` that finally reach the stores at the bottom. To optimize, this funnel can be shrunk both in depth (by reducing redundant extracts and interleaves along paths from loads to stores, or replacing them with more flexible permute operations) and in width (by reducing the number of loads needed). Note that in general such optimizations require the use of permute operations 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 which can exploit parallelism both across loop iterations as well as inside the loop, 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 realize it.

7. Related Work

Several works in the field 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 [18]. Such architectures have better support for reordering vector data than traditional SIMD, and can benefit from advanced compiler technology [19]. 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. The focus of this paper is to show that data reordering can be efficient also for 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]. Reordering within memory operations increases the already long latency of memory operations, may cause a series of cache misses when addressing several remote locations and misses spatial reuse opportunities [31]. Our focus was to analyze and exploit spatial reuse explicitly by the compiler, for relevant platforms that exist today.

In Section 6 we referred to special cases of interleaving that can be vectorized using SIMOMD operations, such as those targeted by [3, 16]. Also in section 6 we discussed in detail the interrelations between our work and SLP [15]. These works do not address the general problem of interleaved data, but the way they handle certain special cases can be incorporated as an extention to our work.

Recent work on classical SIMD [9, 32] focuses on unit-stride accesses, handling the alignment problem which is related to the interleaving problem we deal with. 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 to [28]. To the best of our knowledge, our effort offers the first general treatment of power-of-2 strided accesses that is applicable across classical SIMD platforms.

Vectorization can also be provided by source-to-source optimizers such as VAST [29] (which also handles complex 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 (if accurate performance estimates are used), counter to a general belief [30].

8. Conclusion

We have extended a classic loop-based vectorizer to handle computations that have non-unit stride accesses to data. Counter to a general belief, strided accesses can be vectorized efficiently using standard loop-based SIMD for strides even as large as 16 and 32, provided that the vectorization factor is sufficiently large and that additional operations exists 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.

References

[1] Randy Allen and Ken 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, 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 *PACT*, 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] Aart Bik. *The Software Vectorization Handbook. Applying Multimedia Extensions for Maximum Performance.* Intel Press, 2004.

[6] Jesus Corbal, Roger Espasa, and Mateo Valero. Exploiting a new level of DLP in multimedia applications. In *Intl. Symposium on Microarchitecture*, pages 72–, 1999.

[7] Paul D'Arcy and Scott Beach. StarCore SC140: A new DSP architecture for portable devices. In *Wireless Symposium*. Motorola, September 1999.

[8] K. Diefendorff and P. K. Dubey et al. Altivec extension to PowerPC accelerates media processing. *IEEE Micro*, March-April 2000.

[9] Alexandre E. Eichenberger, Peng Wu, and Kevin O'brien. Vectorization for simd architectures with alignment constraints. In *PLDI*, June 2004.

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

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

[12] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *SIGPLAN Conference on Programming Languages Design and Implementation*, June 1991.

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

[14] 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):589–604, July 2005.

[15] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *PLDI*, 35(5):145–156, June 2000.

[16] J. Lorenz, S. Kral, F. Franchetti, and C. W. Ueberhuber. Vectorization techniques for the blue gene/l double fpu. *IBM Journal of Research and Development*, 49(2/3):437–446, March/May 2005.

[17] Jason Merrill. Generic and gimple: A new tree representation for entire functions. In *the GCC Developer's summit*, June 2003.

[18] Jaime 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*, March 2003.

[19] Dorit Naishlos, Marina Biberstein, Shay Ben-David, and Ayal Zaks. Vectorizing for a simdd dsp architecture. In *CASES*, pages 2–11, October 2003.

[20] Dorit Naishlos and Richard Henderson. Multi-platform auto-vectorization. In *CGO, To appear*, 2006.

[21] Huy Nguyen and Lizy Kurian John. Exploiting SIMD parallelism in DSP and multimedia algorithms using the AltiVec technology. In *Intl. Conf. on Supercomputing*, pages 11–20, 1999.

[22] Diego Novillo. Tree ssa - a new optimization infrastructure for gcc. In *the GCC Developer's summit*, June 2003.

[23] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, pages 43–45, August 1996.

[24] Sabastian Pop, Georges-André Silber, Albert Cohen, Philippe Clauss, and Vincent Loechner. Fast recognition of scalar evolutions on three-address ssa code. Research Report A/354/CRI, CRI/ENSMP, April 2004.

[25] Sebastian Pop, Albert Cohen, and Georges-Andre Silber. Induction variable analysis with delayed abstractions. In *HiPEAC*, November 2005.

[26] Gang Ren, Peng Wu, and David Padua. A preliminary study on the vectorization of multimedia applications for multimedia extensions. In *16th International Workshop of Languages and Compilers for Parallel Computing*, October 2003.

[27] Jaewook Shin, Jacqueline Chame, and Mary W. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures. In *PACT*, September 2002.

[28] Kevin B. Smith, Aart J.C. Bik, and Xinmin Tian. Support for the intel pentium 4 processor with hyper-threading technology in intel 8.0 compilers. *Intel Technology Journal*, 8(1):19–31, February 2004.

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

[30] Crecent Bay Software. Vast/altivec faq: Vectorization for altivec, http://www.crescentbaysoftware.com/altivec_faq.html.

[31] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison Wesley, 1996.

[32] Peng Wu, Alexandre E. Eichenberger, and Amy Wang. Efficient simd code generation for runtime alignment. In *CGO*, March 2005.