Reprinted from the

Proceedings of the GCC Developers' Summit

June 28th–30th, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon Incorporated*
C. Craig Ross, *Linux Symposium*

Review Committee

Ben Elliston, *IBM*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Red Hat*
Gerald Pfeifer, *Novell*
Ian Lance Taylor, *Google*
C. Craig Ross, *Linux Symposium*
Andrew J. Hutton, *Steamballoon Incorporated*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Autovectorization in GCC—two years later

Dorit Nuzman
IBM Haifa Research Lab - HiPEAC Member
dorit@il.ibm.com

Ayal Zaks
IBM Haifa Research Lab - HiPEAC Member
zaks@il.ibm.com

Abstract

The first version of auto-vectorization was contributed to the GCC lno-branch on January 1st, 2004. Later that year it was presented at the second GCC summit, featuring basic capabilities and preliminary experimental results on PowerPC970. Since then, the vectorizer has made a long way, starting from its acceptance to the GCC 4.0 release, and gradually increasing its applicability both in terms of the application domain it can address and the range of platforms it can target.

This paper overviews the evolvement of the vectorizer in the past two years. This includes support for pointer based and unaligned references, conditional operations, reductions, special idioms, type conversions, and a novel generic vectorization of accesses with power-of-2 strides. Some of these features required new abstractions to express vector operations. It took a collaborative effort to devise abstractions that are general enough, applicable to existing architectures, and fit GCC conventions. This collaboration yielded a vectorization scheme that balances the conflicting needs of different platforms while efficiently supporting each individual target. This is the most comprehensive effort that considers the multi-platform aspect of vectorization, demonstrating applicability on diverse SIMD platforms by one compiler. We also present experimental results on a wide range of key kernels and on several different SIMD platforms, and conclude with directions for future work.

1 Introduction

In early June 2004 we introduced the initial implementation of GCC's automatic vectorization optimization at the second annual GCC Developers Summit. At that stage, the vectorizer was still in its infancy on a development branch and was capable of handling simple constructs only: loops with a single basic block that contain unit-stride accesses to aligned memory locations, all accessing data types of the same size, and no loop-carried dependencies. The vectorizer has enhanced constantly in the past two years starting from its acceptance into GCC mainline version 4.0, followed by new features including vectorizing reduction operations (GCC 4.1), reduction pattern recognition (committed to GCC 4.2), and additional enhancements to support non unit stride accesses and multiple types (submitted

to GCC 4.2). In this paper we describe these enhancements and show how collectively they facilitate vectorizing important multimedia kernels. In concert with these functionality enhancements, the vectorizer has also been extended to support many different vector targets [7]. Continued development of the vectorizer takes place on the autovect development branch; the interested reader is referred to the Free Software Foundation website [2] for access to source files. For general background on automatic vectorization, the reader is referred to our previous paper [6].

The main achievements of the vectorizer over the past two years are in three areas:

More loops can now be vectorized, thanks to efficient support for alignment, pointer accesses, conditional operations, reductions, pattern recognition, multiple types and non unit strided accesses.

More platforms are now supported, due to collaborative design of new vector abstractions.

Real performance improvements can now be obtained by vectorizing real world code for a multitude of SIMD platforms.

These achievements provide the first industry-strength framework capable of considering the multi-platform aspect of vectorization [7]. Many users are exercising the vectorizer (see, e.g., [4]), whose patches are now part of official GCC releases (or are pending approval for inclusion in future releases), thereby helping to reveal various issues [5].

The paper is organized as follows. Section 3 explains the major tradeoffs that we faced when introducing new idioms to the intermediate representation of GCC. In section 4 we describe ability of the vectorizer to handle loops with reduction operations which involve loop-carried dependencies. Section 5 shows how the basic infrastructure was seamlessly extended to support non-unit stride accesses to memory, recognizing and exploiting spatial locality efficiently. In section 6 we describe the ability of the vectorizer to handle loops with accesses to multiple data types and sizes, effectively requiring a restricted form of unrolling. Section 7 provides experimental results that exercise the new capabilities and support for different SIMD targets. We present our conclusion in Section 8.

2 Back to the Future

The starting point for this paper is the directions for further development projected two years ago [6]. These were organized into four categories:

*"**Support additional loop forms**. Support for unknown loop bounds and if-then-else constructs is nearly complete. The major remaining restriction on loop form is the nesting level. Vectorization of nested loops will be considered in the future."*

Indeed, the vectorizer now handles counted loops with arbitrary bounds using loop peeling (contributed by Olga Golovanevsky), based on a generic utility to compute number of iterations (contributed by Sebastian Pop and Zdenek Dvorak). An if-conversion pass is also in place prior to vectorization to collapse simple if-then-else constructs (contributed by Devang Patel). Further extension to the if-converter are currently under development, including the handling of loads/stores and collapsing multiple if-then-else constructs. Vectorization of nested loops is yet to be addressed.

*"**Support additional forms of data references**. Potential extensions in this category include en-*

hancements to the dependence tests (as discussed in Section 5) and support for additional access patterns (reverse access, and accesses that require data manipulations like strided or permuted accesses). Exploiting data reuse as in [9] is an optimization related to data references that we plan to consider in the future."

The major enhancement in this category is the support for additional access patterns in the form of non unit strided accesses whose stride is a power of 2, which has been submitted to mainline and is pending review (for inclusion in GCC 4.2; available on the autovect branch). Handling other access patterns such as reverse or permuted accesses has not been addressed yet. The dependence tests have been enhanced to consider dependencies of distance greater than the vectorization factor (as discussed in the above mentioned Section 5), and there have been enhancements to the generic dependence-tests engine (e.g., Omega tests, contribution of Sebastian Pop). The dependence test used by the vectorizer now handles pointers references as well, in addition to the originally supported array references. Using loop distribution to resolve dependencies has not been addressed yet, and neither has the issue of exploiting outer-loop related reuse.

*"**Support additional operations**. Vectorization of loops with multiple data-types and type casting is the first extension expected in this category. This capability requires support for data packing and unpacking, which breaks out of the one-to-one substitution scheme, and cannot be directly expressed using existing tree-codes. The next capabilities to be introduced will be support for vectorization of induction, reduction, and special idioms (such as saturation, min/max, dot product, etc.), using target hooks or adding new tree-code as necessary."*

Much progress has been achieved in this category. Specifically, support for reduction operations (including regular summation, min/max) has been incorporated into GCC 4.1, and support for reduction patterns including dot-product and widening summation has been committed to mainline (for GCC 4.2). Patches that handle multiple data-types and type casting are available on the autovect branch and have been submitted to GCC mainline (pending review). Support for induction is under development.

*"**Other enhancements and optimizations**. Two general capabilities that we are planning to introduce are support for multiple vector lengths for a single target, and the ability to evaluate the cost of applying vectorization. This will require some form of cost modelling for the vector operations. Interaction with other optimization passes should also be examined, and in particular, potential interaction with other (new) passes that might also exploit data parallelism. One example could be loop parallelization (using threads). Another example could be straight-line code vectorization (as opposed to loop based), such as SLP [3]."*

This category of 'other enhancements' is again a subject of future work.

In addition to the above categories, the handling of alignment has been improved considerably over the last two years, with the ability to perform loop versioning (contributed by Keith Besaw) and loop peeling for multiple loads/stores known to have the same misalignment value. Some of the major issues involving the vectorizer have been presented in other forums [7] [8]. This paper provides a summary of the work achieved in the last two years, with additional more recent details not conveyed by prior publications, focusing on mature patches that have been incorporated or at-least submitted to mainline.

The capabilities of the vectorizer two years ago relied on the available idioms of GCC's GIMPLE intermediate representation, and were

limited accordingly. The major enhancements mentioned above involve vector operations that were not previously expressible in GIMPLE. One of the key factors to the success of the recent enhancements was the introduction of appropriate vector abstractions to GCC. This required a collaborative effort of different vendors and individuals, in order to come up with abstractions that are general enough, applicable to existing architectures, and comply with GCC conventions. This collaboration yielded a vectorization scheme that is able to balance the conflicting needs that arise from the diverse nature of SIMD architectures while supporting each individual target efficiently, as we explain next.

3 New vector abstractions

Vector operations are generally represented in GIMPLE like scalar operations: the same operation codes are used, but the arguments are of vector type. 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 are meaningless in the context of scalar computations and are therefore unavailable in GIMPLE. In order to express these mechanisms in the vectorized GIMPLE IL, we introduce some new high-level platform-independent abstractions during the last two years.

The general issues and considerations involved in introducing new vector operations to GCC and the GIMPLE IL are discussed in detail in [7], especially in the context of alignment and reduction mechanisms. These considerations include: (1) weighing the benefits of compound abstractions with the advantages of using simpler more basic abstractions; (2) balancing the generality of abstractions with the applicability of existing architectures; (3) considering existing GCC conventions, and; (4) performance considerations — striving to use abstractions that translate into the most efficient instructions available on targets.

For example, consider the issue of non unit stride accesses. Most SIMD architectures provide access only to contiguous memory items, from base addresses that are aligned on a natural vector size boundary. Computations, on the other hand, may access data elements in an order which is neither contiguous nor adequately aligned. Special data reordering mechanisms are provided to help cope with such situations. These mechanisms usually involve additional memory accesses and shuffling instructions for combining data elements from different vectors. The vectorizer must be aware of the available data reordering mechanisms in order to determine whether vectorization of a given computation on a given platform is possible and profitable. It also needs to generate code that correctly and efficiently accesses data located at disjoint and potentially unaligned memory addresses.

We therefore need to abstract low-level data shuffling and alignment handling constructs, and express them in the GIMPLE IL. However, these data shuffling mechanism often differ widely from one SIMD platform to another, posing a challenge to formulate adequate abstractions. A similar situation involving data shuffling arises when vectorizing computations that contain type conversions. In such cases, data elements that reside in one vector need to be expanded and placed in two or more vectors, and vice-versa.

Data shuffling can be accomplished using a general "`permute`" operation, which selects an arbitrary set of elements from two vectors (possibly restricted to only one) and packs

them in one vector. Most platforms, however, do not support such a powerful permute idiom in its most general form. Only the AltiVec `vperm` instruction allows arbitrary, variable permutation. Other platforms (MMX, SSE, MIPS, IA-64 and SPE) either have instructions to merge the high or low halves of two vectors, or something akin to the AltiVec `vperm` that accepts fixed (compile-time constant) permutations. This is a perfect example of the conflict between the desire to introduce general, powerful abstractions and the need to consider what the target platforms actually support. Instead of using a general permute abstraction, the different flavors of data shuffling mechanisms are more appropriately addressed in GCC via simpler, specialized idioms that better match the available technology: the `vec_extract_even` and `vect_extract_odd` abstractions extract elements at even/odd indices, respectively, from two vectors, treated as one stream of elements; the `vec_interleave_hi` and `vec_interleave_lo` abstractions extract the high/low-order elements of two vectors, respectively, and merge them together. These simple abstractions are supported efficiently on most SIMD targets, and provide the means to handle power-of-2 strided accesses [8], as explained in Section 5.

A similar case of data shuffling is related to handling accesses to unaligned addresses, where data elements are to be extracted from two vectors according to the misalignment of the address. Here too, instead of using a general permute abstraction, we created the specialized `realign_load` idiom [7] which takes three arguments: two vectors and a `realignment_token`. The `realignment_token` can be an address, a bit mask, a vector of indices, an offset, or anything that can be generated as a function of the respective address. The `realignment_token` hides low-level details, allowing each target to express its best alignment handling capabilities thereby keeping the `realign_load` idiom general enough yet not too general.

4 Reduction

The basic support for vectorizing loops with reduction operations is provided in GCC version 4.1. Enhanced support for more complex reduction patterns was committed to mainline and will become part of GCC version 4.2.

4.1 Basic reduction

A loop containing a basic reduction operation is depicted in Figure 1(a), where `op` is an associative and commutative operation (such as addition or min/max) which creates a cross iteration data flow dependence cycle, formed by the reduction variable (`a` in the example). If there are no other uses of `a1` and `a2` in the loop, and if it's ok to change the computation order, the vectorizer transforms code by having the loop compute a vector of partial results followed by a loop epilog which reduces this vector to the desired scalar result, thereby altering the order of operations. This is shown in Figure 1(b), where the loop executes parallel independent `op` operations (`vop`) and the loop epilog (labelled `loop_exit`) reduces the vector of partial results using a `reduc_op`, from which the final scalar element desired is extracted (`bit_field_ref`).

Specific details involving epilog code generation and accumulator initialization are described by Nuzman and Henderson [7].

(a) scalar :

```
loop:
      a1 = phi <a0, a2>
s1:   x = ...
s2:   a2 = op <x, a1>

loop_exit:
      a3 = phi <a2>
s3:   use <a3>
s4:   use <a3>
```

(b) vector :

```
loop:
      va1 = phi <va0, va2>
vs1:  vx = ...
vs2:  va2 = vop <vx, va1>

loop_exit:
      va3 = phi <va2>
vs3:  va4 = reduc_op <va3>
vs4:  a5=bit_field_ref<va4,0>
s3:   use <a5>
s4:   use <a5>
```

Figure 1: Basic reduction

4.2 Reduction patterns

Vector targets often have efficient support for complex operations that involve reduction operations. For example, a target may provide an instruction that both multiplies two vectors of multipliers and adds (some of) the products together, to support efficient dot product computations. This example also helps to deal with multiple types (see Section 6), as fewer results of a wider type need to be recorded. It is therefore important for the vectorizer to detect when such complex operations can be applied.

We implemented a pattern recognition engine (vect_pattern_recog) for this purpose, which has been committed to mainline (planned for GCC version 4.2). This engine searches for a sequence of statements that follows a certain idiom; if such a sequence is found, a new 'pattern' statement `ps` representing the idiom is added before the last statement of the sequence `ls` (with cross links between the two), and `ls` is marked as 'in_pattern'. Later on, if we reach `ls` during the bottom-up scan to mark statements relevant for vectorization (vect_mark_relevant), we mark the new `ps` statement instead of `ls`. This way we detect if all intermediate statements of the sequence are used only by `ls`, and if so vectorize only `ps`.

```
sum0 = phi <init, sum1>
dx = (type1) x;
dy = (type1) y;
dprod = dx * dy;
[dprod = (type2) dprod;]
sum1 = dprod + sum0;
```

Figure 2: Reduction pattern example: dot product

The original support for reduction operations handles single statements of the form `a = op <x, a>` having two arguments, where the first (`x`) is defined in the loop and the second (`a`) is the reduction variable defined by the loop-header ϕ, and both have the same type. The extension to handle reduction patterns considers more than one `x` argument defined in the loop, keeping the last `a` argument as the reduction variable, and allowing the type of `a` to be wider than the types of the `x` arguments. This extension captures the dot-product and widening-summation reduction patterns.

The pattern for dot product, for example, is given in Figure 2, where `dx` is double the size of `x`, `dy` is double the size of `y`, `dx`, `dy`, `dprod` all have the same type, `sum` is the same size of `dprod` [or wider], and `sum` has been recognized as a reduction variable. The pattern statement for such dot product patterns is denoted by a new DOT_PROD_EXPR tree-code taking three arguments `<x,y,sum0>`. It is equivalent to a WIDEN_MULT_EXPR statement of `<x,y>` (see Section 6) followed by a regular PLUS_EXPR or a WIDEN_SUM_EXPR state-

ment.

When vectorizing reduction patterns that involve multiple types, care must be taken to select the appropriate types in the final loop epilog code; inside the loop we use statements with complex tree-codes such as DOT_PROD_EXPR, but at the epilog we may use the simple PLUS_EXPR statements to combine the partial results together, using both the wider type and the tree-code of the original scalar reduction operation. This is contrary to simple reductions in which the types of all arguments (including that of the reduction variable) are the same, allowing us to use the same vector type and tree-code for the epilog code and for the code inside the loop.

5 Non Unit Stride Accesses

Work on vectorizing loops which access data in a non-consecutive strided pattern has been presented recently by Nuzman, Rosen and Zaks [8]. This work has been submitted to GCC mainline for inclusion in version 4.2, and is pending review (at the time of writing; contribution of Ira Rosen). The challenge in vectorizing accesses to non-consecutive addresses is twofold: first, the appropriate sets of loads/stores and extract/interleave instructions have to be established, and second, spatial reuse opportunities should be identified and exploited. In terms of the underlying infrastructure, a new type of analysis which groups together independent statements was devised to represent spatial reuse partners. In contrast, previous analyses dealt with individual data references, dependent statements or the entire loop.

Despite the above challenges, integrating the support for non-unit stride accesses with the existing framework was pretty smooth. We extended the existing dependence resolution traversal over pairs of load/store statements, to construct groups of interleaved loads or stores that have the same stride and adjacent base addresses. The data is then provided to each group using a set of loads and extract_even/odd operations, or a set of interleave_low/high operations and stores. This mechanism supports strides that are a power of 2, which are the more common strides and are efficiently handled on most platforms using these abstractions.

For example, if we have a group of four loads with stride four and consecutive base addresses, and VF=8, we will generate the following instructions: a set of four vector loads that load consecutive elements into vectors (0...7), (8...15), (16...23), (24...31) followed by a set of four (intermediate) extract_even/odd operations that produce the vectors (0,2...14), (16,18...30), (1,3...15), (17,19...31) and a set of four (final) extract_even/odd operations that produce the desired vectors: (0,4...28), (1,5...29), (2,6...30), (3,7...31). The case of strided stores is analogous, using interleave_lo/hi instead of extract_even/odd. Additional examples and details are available in previous publications [8, 1].

6 Multiple Types

When the computations inside a loop operate on data-types of different sizes, the vectorization scheme can no longer be a simple 'strip-mine by a single vectorization factor and replace 1-to-1' [6]. This is because different operations prefer different vectorization factors. In general, we set the vectorization factor according to the smallest data type in the loop; operations that operate on larger data types will be replicated similar to loop unrolling. Peeling the loop to align accesses was also updated to

consider multiple data types; the original misalignment values in byte units had to be augmented with the corresponding data type size, to deduce the correct number of elements to handle in the loop prolog.

The overall transformation scheme is as follows. First, as mentioned above, we set the vectorization factor (VF) to the largest value according to the smallest data type in the loop. Then when we vectorize a statement that operates on a data-type of which VF elements cannot fit in one vector word, multiple vector statements are generated to compute VF results. For example, say a loop contains (1 byte) chars and (4 byte) ints, and the vector size (VS) is 16 bytes; the VF in this case will be 16 due to the operations on chars. Each scalar statement that deals with chars is vectorized as usual by replacing it with its vector counterpart. In contrast, each scalar statement that deals with ints is replaced by four vector statements (denoted VS1.0, VS1.1, VS1.2, VS1.3) that together compute 16 results in one iteration of the vectorized loop (see Figure 3). This is because only four ints fit in one vector word. When we continue and vectorize the statement that uses the value defined by S1 (denoted S2), each of the 4 vector statements that we generate (denoted VS2.0, VS2.1, VS2.2, VS2.3) will use the respective vector value defined by VS1.0, VS1.1, VS1.2, VS1.3. In order to be able to find these 4 vector definitions, we chain the vector statements together (via the `RELATED_STMT` field of the `stmt_info` struct: VS1.0→ VS1.1→ VS1.2 → VS1.3. For this reason a `stmt_info` struct is now created for the newly generated vector statements as well). The changes to the vector transformation routines therefore mainly consist of adding a loop around the original vector-statement-creation code to create multiple vector statements per scalar statement, and perform the bookkeeping described above to chain together these vector statements.

(a) scalar :

```
S1:     x = memref
S2:     z = x + 1
```

(b) after vectorizing S1 :

```
VS1.0:  vx0 = memref0
VS1.1:  vx1 = memref1
VS1.2:  vx2 = memref2
VS1.3:  vx3 = memref3
S1: x = memref
S2: z = x + 1
```

(c) after vectorizing S2 :

```
VS1.0:  vx0 = memref0
VS1.1:  vx1 = memref1
VS1.2:  vx2 = memref2
VS1.3:  vx3 = memref3
S1: x = memref
VS2.0:  vz0 = vx0 + v1
VS2.1:  vz1 = vx1 + v1
VS2.2:  vz2 = vx2 + v1
VS2.3:  vz3 = vx3 + v1
S2: z = x + 1
```

Figure 3: Multiple types: simple example

We now describe several issues that arise for certain kinds of computations operating on multiple data types. Many of these issues are related to loop unrolling in general, but are raised here in the context of vectorization.

6.1 Multiple load/stores

When vectorizing a load/store statement and replacing it with multiple vector load/store statements, one issue that comes up is the creation of the vector pointer: we always create a single vector pointer per scalar load/store that we vectorize; all copies of the corresponding vector loads/stores that we generate use the same vector pointer, which is bumped by VS bytes between each pair of consecutive vector loads/stores.

An alternative approach is to bump the mutual vector pointer only once in each iteration of

the vectorized loop, and use a different offset for each of the multiple vector load/store statements involved. It is possible for a subsequent pass to introduce such multiple offsets (which may require more registers) in order to reduce dependencies.

6.2 Multiple reductions

One issue that comes up when generating multiple vector statements for a scalar reduction statement, is how to combine the multiple accumulators. Suppose we vectorize a summation of ints in a loop that also operates on chars, and the VS is 16 bytes. We generate four vector statements to add 16 int elements in each iteration of the vectorized loop. One option is to use four independent accumulators, and combine them at the loop epilog (Figure 4(b)); another option is to have each of the four vector statements feed the next, effectively using a single accumulator (Figure 4(c)).

We chose to implement the latter option of the single accumulator primarily because it uses fewer registers. A subsequent accumulator-expansion pass could in the future replace the single accumulator with multiple accumulators, similar to GCC's current modulo-variable-expansion optimization in the loop unroller (except that in our case we unroll the relevant statements ourselves). In contrast, the converse operation of collapsing multiple independent accumulators feeding a final reduction at the loop epilog into a single accumulator, primarily to save registers, seems more difficult to recognize and realize.

(a) scalar :

```
x = memref
z = z + x
```

(b) multiple vector accumulators :

```
loop:
  vz0 = phi (init0, vz0)
  vz1 = phi (init1, vz1)
  vz2 = phi (init2, vz2)
  vz3 = phi (init3, vz3)
  vx0 = memref0
  vx1 = memref1
  vx2 = memref2
  vx3 = memref3
  vz0 = vz0 + vx0
  vz1 = vz1 + vx1
  vz2 = vz2 + vx2
  vz3 = vz3 + vx3
epilog:
  vz3 = (vz0 + vz1 + vz2 + vz3)
```

(c) single chained vector accumulator :

```
  vz = phi (init, vz)
  vx0 = memref0
  vx1 = memref1
  vx2 = memref2
  vx3 = memref3
  vz = vz + vx0
  vz = vz + vx1
  vz = vz + vx2
  vz = vz + vx3
```

Figure 4: Multiple types: reduction

6.3 Type demotion

When operations involving different data types are not independent, data is transferred from one type to another. Type demotion refers to

the case where data is transferred from a larger type to a smaller type. In this case, one usually either disregards the excessive bits (modulo demotion) or uses them to perform saturation if needed (saturating demotion). We added new tree-codes to support both options, namely `VEC_PACK_MOD_EXPR` and `VEC_PACK_SAT_EXPR`, and accompanying optabs. The term 'pack' stems from the fact that we can place more elements in a vector word after demoting them.

The current implementation of type-demotion is restricted to 'half-demotion' cases where the wider-type (the type of the arguments) is twice that of the smaller type (the type of the result). Note that most targets provide instructions that directly support half-demotion, where the contents of two source operands are packed into a single destination operand; for this reason, a scalar half-demoting operation may be replaced by a single vector statement (which is fed by multiple vector statements).

6.4 Type promotion

Type promotion refers to the case where data is transferred from a smaller type to a larger type. This often occurs when an operation produces a result that can be larger than its operands, the prominent case being multiplication. In general, type promotion appears as a cast operation; the operands are first casted and then the operation is performed on the wide data type. However, most targets provide vector support that combines certain operations with type promotion. It is important for the vectorizer to make use of such combined operations because of their efficiency and because targets may not support the same operation on wider operands.

In addition to combining operations with type promotion, targets typically provide instructions that produce only a subset of the results, each result being wider than the input arguments of the operation. In some cases, as in the case of widening multiplication, the subset of products may be noncontiguous, requiring subsequent shuffling statements to sort the obtained products according to the original (multiplier) order, if needed. Again, the vectorizer should identify when such shuffling code needs to be generated.

Type promotion involves taking one vector of elements (or several, as in the case of widening multiplication) and producing a vector in which each element is of larger size; the resulting vector therefore typically does not fit in a single vector register, so several vector statements are used to replace a single scalar statement (chained together as described earlier in this section).

We implemented support for both general cast promotion and for widening multiplication. General cast promotions are represented using the existing `NOP_EXPR` tree-code, and are vectorized using new `vec_unpack_hi/lo` idioms. Widening multiplication promotions are represented using the new `WIDEN_MULT_EXPR` tree-code, and are vectorized using the `vec_widen_mult_hi/lo` idioms, which preserve the order of the products. If the vectorizer can prove that the order of the products does not have to be preserved (e.g., when the products are used only to feed a reduction computation) then the target hooks builtin_mul_widen_even/odd are used if available; they produce the products of even and odd elements in two separate vectors.

The current implementation of type-promotion is restricted to 'double-promotion' cases where the wider-type (the type of the result) is twice that of the smaller type (the type of the arguments). Note that most targets provide instructions that directly support double-promotion, where half of the elements in the source

operand(s) are expanded to fill the destination operand; for this reason, scalar double-promoting statement are often replaced by pairs of vector statements.

The following new tree-codes and accompanying optabs were added to express vectorized widening operations. To double-promote the low/high half of a vectors elements, using sign-extension for signed and zero-extension for unsigned data types: `VEC_UNPACK_[LO, HI]_EXPR` To produce double-width products of the low/high half of (signed/unsigned) vector multipliers and multiplicands: `VEC_WIDEN_MULT_[LO,HI]_EXPR`.

The `TARGET_VECTORIZE_BUILTIN_MUL_WIDEN_[ODD,EVEN]` target hooks were added, to produce double-width products of odd/even half of vector multipliers and multiplicands. These hooks are used only when we know that the order of products can be altered. We currently detect such situations during the '`mark_stmts_to_be_vectorized`' scan, which was augmented to indicate if a statement is used (only) by reduction operations (in addition to the original indication if it is used in the loop or not).

7 Experimental Results

The results we present in this section exemplify how the enhancements developed in the past two years can be put to work, and successfully vectorize important kernels on a multitude of platforms. We generated the results automatically using the autovect-branch which contains the enhancements of GCC 4.1 and those submitted to GCC 4.2, available from [2]. Experiments were performed on an IBM PowerPC970 processor with Altivec, an AMD Athlon processor with SSE2, an Intel PentiumD (dual-core Pentium 4) processor with SSE2, an Intel Itanium2, and a MIPS64 instruction-accurate simulator with paired-single-fp support. For the AMD/Intel and MIPS platforms, the measurements as well as the required platform-specific development were performed by Richard Henderson and Chao-Ying Fu, respectively.

Name	Description
saxpy_fp	constant times a vector plus a vector
sdot_fp	dot product of two vectors
vecsum_fp	sum elements of a vector
vecmax_fp	find maximum over elements of a vector
cond_replace_fp	copy selective vector elements
add_sat_fp	add two vectors and clip the result
tcpip_checksum_s16	tcpip checksum
audio_dissolve_s16	audio steam fade away
dot_s16	dot product (used in audio FIR filters)
eudist_s16	euclidian distance of vectors (GSM EFR)
linear_comb_s16	linear combination of two vector
vecmax_s16	find maximum over elements of a vector
video_dissolve_u8	image fade away
linear_comb_u8	linear combination of two vector
vecsum_u8	summation of a vector elements
vecmax_u8	find maximum over elements of a vector
i2_cxdot-fp	complex dot product
i4_mixStreams-s16	mix two 4-channel audio streams
i8_cvt_codec-u8	convert rrggbbaa codec to aarrggbb

Table 1: Benchmark description

Table 1 provides a brief description of the kernels used in our experiments. The kernels we use are representative of the main computations in important applications from different domains—linear algebra, video and audio processing, and networking and cover all the features discussed in previous sections (if-conversion, multiple types, reduction, and reduction patterns). The last set of kernels also includes interleaved data (strided accesses), with strides 2, 4, and 8 in `i2_cxdot`, `i4_mixStreams`, and `i8_cvs_codec`, respectively. We report the speedup factors achieved by an automatically vectorized version over the sequential version of the benchmark, compiled with the same optimization flags. Time is measured using the `getrusage` routine (except for MIPS speedups that measure dynamic instruction count instead) and includes any overheads introduced by vectorization. Figures 5–9 summarize the speedup factors obtained for

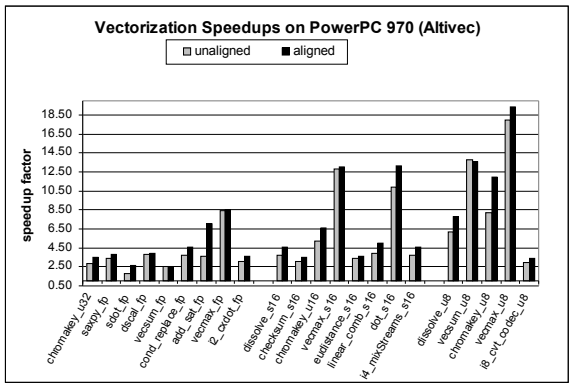

Figure 5: PowerPC970 (Altivec) speedups

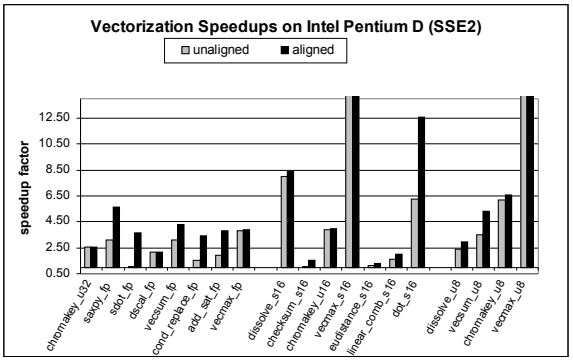

Figure 7: PentiumD (SSE2) speedups

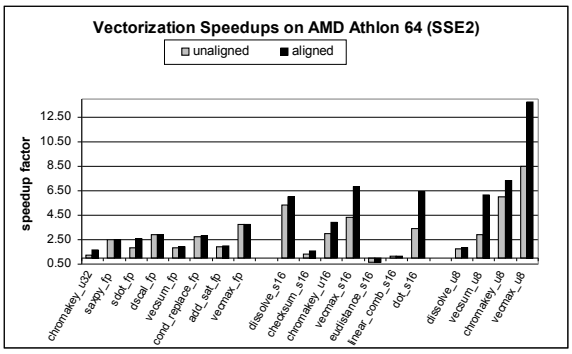

Figure 6: Athlon (SSE2) speedups

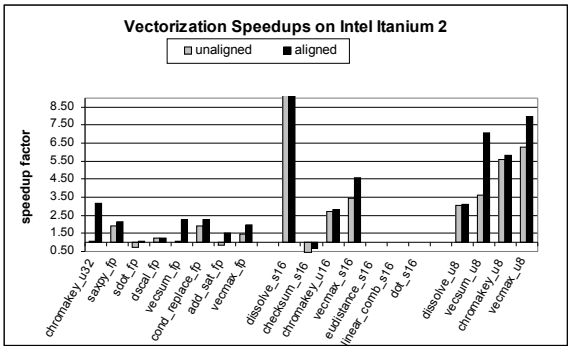

Figure 8: Itanium2 speedups

two versions of each testcase—one in which the alignment of the data is unknown and the other when the data is aligned.

PPC970 Speedups (Altivec): On PowerPC970 we expect an improvement factor between 2 – 4 on floating point benchmarks (one SIMD unit vs. two scalar units), speedups between 4 – 8 on the short benchmarks, and between 8 – 16 on the char benchmarks (minus realignment overhead on the unaligned versions). The speedups obtained are generally within these ranges. Super-linear speedups on dot_s16, vecmax_u8/s16/fp, add_sat_fp, and cond_replace_fp are due to the availability of specialized instructions on the vector unit, while they are absent from the scalar unit

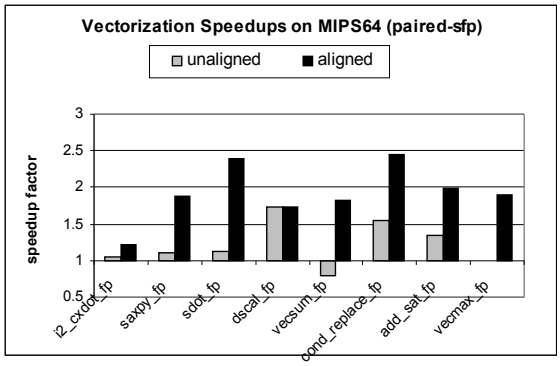

Figure 9: MIPS64 speedups

(`mulsum` (dot), `max`, and conditional instructions). Lower speedups on dissolve_s16/u8, checksum, and linear_comb are due to additional overheads incurred by type conversion and widening multiplication, which take twice as many vector operations to accomplish. Lower speedups on i2_cxdot, i4_mixStreams, and i8_cvs_codec are due to data reordering overheads (usage extracts nad interleaves) to cope with interleaving levels (strides) 2, 4, and 8, respectively.

x86 (Athlon and Pentium) Speedups (SSE2): While the SSE2 architecture supports 128-bit vectors, the current implementations can only operate on 64-bit simultaneously. Therefore, the expected improvement on the benchmarks is between `VF`$/2$ and `VF`, closer to `VF`$/2$ (2 for the fp benchmarks, 4 for the short benchmarks, and 8 for the char benchmarks), minus alignment handling overhead on the unaligned versions. The super-linear speedup on vecmax_s16/u8and dot_s16 is due to inefficient code that is generated for the scalar version.

Itanium2 speedups: We expect an improvement factor of 2 for floats, as the SIMD float instructions run on the same functional units as the scalar float instructions. For integer data, we expect the improvement to be correlated with the number of elements packed in the word size. The high speedup in dissolve_s16 is due to the slower 32-bit integer multiply used in the scalar version as opposed to the parallel 16-bit multiply used for dissolve_u8. The eudist_s16, linear_comb_s16 and dot_s16 do not get vectorized on this platform due to the lack of a 32-bit vector integer multiply.

MIPS64 Speedups: The latencies on the paired-single-fp instructions for MIPS64 are similar to the latencies of scalar instructions, and both share the single fp functional unit. The expected speedup is therefore `VF` $= 2$ (minus the alignment handling overhead). Super-linear improvements on dot_fp and cond_replace_fp are due to inefficient addressing in the scalar code compared to the vector code, and lack of conditional move in the scalar unit, respectively. Lower speedups on i2_cxdot are due to data ordering overheads to handle a strided access.

8 Conclusions

The auto-vectorization optimization of GCC has been enhanced in the past two years in three main areas: (1) adding support to vectorize more complex computations involving in particular reductions, multiple types and non unit strides; (2) extending the GIMPLE intermediate language to represent the desired vector abstractions, through active collaboration that addressed the needs of different platforms and resulted in acceptance into GCC's mainline code base; and (3) applying these stable functional enhancements to important kernels on a multitude of platforms to achieve significant performance improvements. More work lies ahead, including items suggested two years ago and refinements of currently supported features.

9 Acknowledgments

Many people have contributed to the vectorizer over the past two years. We tried to specify major individual contributions throughout the paper. Richard Henderson has reviewed our patches, and together with Ira Rosen, Daniel Berlin, Sebastian Pop, and Zdenek Dvorak have provided continuous support; thanks to all!

References

[1] Free Software Foundation. Auto-Vectorization in GCC,

http://gcc.gnu.org/projects/tree-ssa/vectorization.html.

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

[3] 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.

[4] T. Moene. GFortran: Compiling a 1,000,000+ line NumericalWeather Forecasting System. In *Proc. of the GCC Developers Summit*, pages 159–164, June 2005.

[5] T. Moene. Public and private communication. 2005 and 2006.

[6] D. Naishlos. Autovectorization in GCC. In *Proc. of the GCC Developers Summit*, pages 105–117, June 2004.

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

[8] D. Nuzman, I. Rosen and A. Zaks. Auto-Vectorization of Interleaved Data for SIMD In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2006.

[9] 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.