Reprinted from the

Proceedings of the GCC Developers' Summit

July 18th–20th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

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

Proceedings Formatting Team

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

Loop-Aware SLP in GCC

Ira Rosen Dorit Nuzman Ayal Zaks
IBM Haifa Labs – HiPEAC Member
{irar,dorit,zaks}@il.ibm.com

Abstract

The GCC auto-vectorizer currently exploits data parallelism only across iterations of innermost loops. Two other important sources of data parallelism are across iterations of outer loops and in straight-line code. We recently embarked upon extending the scope of auto-vectorization opportunities beyond inner-loop inter-iteration parallelism, in these two directions. This paper describes the latter effort, which will allow GCC to vectorize unrolled-loops, structure accesses in loops, and important computations like FFT and IDCT (as well as several testcases in missed-optimization PRs).

Industry compilers like icc and xlC already support SLP-like vectorization, each in a different way. We want to introduce a new approach for SLP vectorization in loops, that leverages our analysis of adjacent memory references, originally developed for vectorizing strided accesses. We extend our current loop-based vectorization framework to look for parallelism also within a single iteration, yielding a hybrid vectorization framework. This work also opens additional interesting opportunities for enhancing the vectorizer—including partial vectorization (right now it's an "all or nothing" approach), permutations, and MIMD (Multiple Instructions Multiple Data, as in subadd vector operations such as in SSE3 and BlueGene). We will describe how SLP-like vectorization can be incorporated in the current vectorization framework.

1 Introduction

SIMD (Single Instruction Multiple Data) capabilities have become an essential part of today's processors, both general-purpose and embedded. This continues to hold with the recent trend to build hybrid architectures, like the Cell/B.E. [3], that combine multiple specialized vector-only processing cores alongside the main CPU. Compilers have, therefore, been extended to automatically extract data parallelism in programs and translate that into vector operations.

Since most opportunities for data parallelism usually occur in loops, and more specifically across iterations of a loop, classic vectorization techniques have traditionally focused on exploiting exactly that kind of parallelism ([4], [24]). This is also the kind of parallelism that GCC vectorizer has been originally designed to target ([13], [15]).

However, there are many kinds of important computations that cannot be vectorized if considering only the context of the loop. Figure 1a shows, for example, an unrolled loop, hand-optimized for a sequential machine, which is typical for multimedia kernels [9]. Other possible examples include structure accesses, such as RGBA in image processing (see Figure 1b), non-SIMD kind of parallelism, like subadd pattern, which is popular in complex computations (see Figure 1c), and numerical computations, like FFT, that require data reorganization (see Figure 1d).

In order to vectorize such computations, the classic loop-based approach to vectorization needs to be extended to look into other sources of data parallelism, beyond loop iterations. The Superword Level Parallelism (SLP) approach to vectorization ([9]) does exactly that: it 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. It also has several limitations that we discuss later in the paper; however, these can be mitigated by combining the SLP vectorization technique with the loop-based vectorization technique.

This paper describes how we use the SLP approach for vectorization to extend and improve the loop-based vectorization in GCC, building on our strided accesses vectorization infrastructure ([16], [17]).

(a) unrolled:

```
do {
  dst[0] = (src1[0] + src2[0]) >> 1;
  dst[1] = (src1[1] + src2[1]) >> 1;
  dst[2] = (src1[2] + src2[2]) >> 1;
  dst[3] = (src1[3] + src2[3]) >> 1;

  dst += 4;
  src1 += 4;
  src2 += 4;
}
while (dst != end);
```

(b) structure accesses:

```
for (i = 0; i < n; i++) {
  tmp_r = image_in[i].r * fade;
  image_out[i].r = tmp_r >> 8;
  tmp_g = image_in[i].g * fade;
  image_out[i].g = tmp_g >> 8;
  tmp_b = image_in[i].b * fade;
  image_out[i].b = tmp_b >> 8;
  tmp_a = image_in[i].a * fade;
  image_out[i].a = tmp_a >> 8;
}
```

(c) non-SIMD:

```
for (i = 0; i < n; i++) {
  tmp1 = a[i].real * b[i].real;
  tmp2 = a[i].imag * b[i].imag;
  tmp3 = a[i].imag * b[i].real;
  tmp4 = a[i].real * b[i].imag;

  c[i].real = tmp1 - tmp2;
  c[i].imag = tmp3 + tmp4;
}
```

(d) permutations:

```
for (i = 0; i < n; i++) {
  c[4i] = a[4i] + b[4i+1];
  c[4i+1] = a[4i+2] + b[4i+3];
  c[4i+2] = a[4i+3] + b[4i];
  c[4i+3] = a[4i+1] + b[4i+2];
}
```

Figure 1: Examples that require SLP-style of vectorizaion

The rest of the paper is organized as follows. Section 2 provides the conceptual and high-level motivation, considerations, and general approach to the problem of combining these two vectorization approaches. In Section 3 we give a brief overview of the current vectorization pass in GCC, and in Section 4 we describe in detail the technical and engineering steps that are necessary in order to extend the vectorizer in the direction we described above. Section 5 describes the current status of this project and plans for future work. Section 6 discusses related work. We present our conclusions in Section 7.

2 From Vectorization of Interleaved Data to Loop-aware SLP

The GCC auto-vectorizer follows the classic approach for vectorization, which focuses on loops, and attempts to detect data parallelism across different iterations of the loop. After some general properties of the entire loop are analyzed (e.g., data dependencies), each statement is analyzed and vectorized independently of the other statements in the loop. For each statement, the vectorizer tries to group together `VF` occurrences of that statement from `VF` different consecutive iterations, where `VF` is the *Vectorization Factor*—the number of data elements to be operated on in parallel. This is what we refer to as a "1–1 replace" approach—each scalar statement is mapped to (a conceptually) one-vector statement that performs the respective operation on `VF` data elements from `VF` consecutive iterations of the loop.

Last year GCC was extended to handle non-consecutive memory accesses whose access pattern is strided, with power-of-2 strides. Such access patterns occur, for example, when operating on complex data in which the real and imaginary data elements are interleaved (in this case the stride would be 2) (see Figure 1c), or when operating on images whose pixels are represented as interleaved RGBA elements (in which case the stride would be 4) (see Figure 1d), etc. Adding this new capability to the GCC auto-vectorizer meant extending the classic loop-based single-statement-at-a-time approach in the following way: a new analysis pass that detects groups of interleaved memory accesses was added. In order to avoid redundant vector memory accesses and exploit data reuse, these groups of memory references are transformed together. This way, a group of scalar memory references that accesses a range of δ interleaved data elements, are considered together when transforming them into a group of vector memory references that accesses $(VF * \delta)$ data elements, along with vector data rearrangement operations. In the case of interleaved loads, the vector loads are followed by a sequence of `extract_even/odd` vector operations that de-interleaves the loaded data into separate vectors that can be operated upon in parallel. In the case of interleaved stores, the vector stores are preceded by a

sequence of `interleave_high/low` vector operations that interleave the data elements together into vectors that are then stored to memory. See [16] for more details.

(a) scalar:

```
for (i = 0; i < n; i++) {
  t1 = b[2i];
  t2 = b[2i+1];
  a[2i] = t1;
  a[2i+1] = t2;
  c[i] = C;
}
```

(b) SLP:

```
for (i = 0; i < n; i++) {
  vt = b[2i:2i+1];
  a[2i:2i+1] = vt;
  c[i] = C;
}
```

(c) loop-based:

```
vc = {C, C};
for (i = 0; i < n; i += 2){
  vb1 = b[2i:2i+1];
  vb2 = b[2i+2:2i+3];

  vt1 = extract_even(vb1, vb2);
  vt2 = extract_odd(vb1, vb2);
  va1 = interleave_high(vt1, vt2);
  va2 = interleave_low(vt1, vt2);

  a[2i:2i+1] = va1;
  a[2i+2:2i+3] = va2;

  c[i:i+1] = vc;
}
```

(d) loop-aware SLP:

```
vc = {C, C};
for(i = 0; i < n; i+=2) {
  vb1 = b[2i:2i+1];
  vb2 = b[2i+2:2i+3];

  a[2i:2i+1] = vb1;
  a[2i+2:2i+3] = vb2;

  c[i:i+1] = vc;
}
```

Figure 2: Comparison of the different vectorization approaches: SLP, loop-based, and combined

The important thing to notice for the purpose of this paper, is that the single-statement-at-a-time approach ("1–1 replace") is extended to a single-**group**-at-a-time approach, in which some trivial groups are of size 1 (when there is no interleaving, such as the access to array `c` in Figure 2a), and some groups are of size δ (such as the access to array `b` in Figure 2a, where $\delta = 2$, and for which we perform a "δ-δ replace"). Note that we still continue to combine `VF` different iterations when vectorizing, hence the trip-count in the vectorized loop is reduced by a factor `VF`. This is illustrated in Figure 2c, which shows how the current loop-based vectorization scheme vectorizes the scalar loop shown in Figure 2a, assuming $VF = 2$.

The SLP vectorization approach on the other hand groups VF statements from the **same** iteration into a vector statement. It looks at flat code sequences (or flattened code sequences in case if-conversion is applied beforehand—see [21]). So it is in fact not aware of the loop context, and can be applied to basic-blocks anywhere in the program. The SLP approach starts by looking for groups of accesses to adjacent memory addresses, attempting to pack them together into vector load/store operations. These groups of adjacent memory references are used as the seed to an analysis that, starting from this seed, follows the def-use chains between statements, in search for computation chains that can be vectorized. Figure 2b shows how the SLP approach would vectorize the loop in Figure 2a.

Comparing the two approaches, as illustrated in Figure 2, there are several differences that can be seen. Since the current (naive) loop-based approach considers each group of accesses separately, when it vectorizes the `a`-references it does not consider how they are used, and does not see that in fact the loaded `a`-elements can be operated on in parallel without de-interleaving them into separate vectors `vt1`, `vt2` (data reorganization is required only if the computation actually performed different operations on `t1` and `t2`). This results in the redundant extract/interleave computations in Figure 2c. While the SLP approach vectorizes the `a,b`-accesses in the example more efficiently, it cannot vectorize the `c`-access without considering the context of the loop and unrolling it ahead of time such that sufficient (`VF`) occurrences of the store to array `c` are created so that they can be packed into a vector statement. The loop-based approach on the other hand, does "see" the potential parallelism across different iterations and can use that to vectorize the store to array `c`.

There are yet additional tradeoffs that the example Figure 2 illustrates, which we summarize below. But it is already evident that the natural next step, is to combine the two approaches into a single framework, and

consider both intra and inter-iteration parallelism simultaneously, while enjoying the advantages of both approaches. From the SLP approach we borrow the def-use analysis within an iteration to fix the current deficiency of our loop-based approach when vectorizing a group of adjacent memory-accesses that don't require de-interleaving. From the loop-based approach we borrow the loop-aware ability to consider different unrolling factors, as well as the potential to amortize costs across loop iterations, and handle more elaborate seeds (e.g., groups of memory accesses with gaps), as explained below. Figure 2d shows the resulting vectorized code when performing SLP-vectorization while taking advantage of the loop context. Or, an alternative way to view Figure 2d: the resulting vectorized code when performing loop-based vectorization extended to consider intra-iteration behavior.

To conclude this section, we want to summarize the tradeoffs that are considered when combining the above two approaches. Loop-aware SLP improves upon the classic SLP approach in the following ways:

1. By focusing on loops we limit SLP to the code portions that are more likely to impact the performance of the program.

2. Applying SLP in the context of a loop optimization pass allows us to optimize the generated code across loop iterations and amortize vectorization overheads. One trivial example is set-up of vector registers, such as the initialization of vector `vc` (which is done out of the loop). It is not always as trivial to move out such invariant computations out of the loop. Another example is the handling of misaligned accesses. When considering the loop as a whole, it can be transformed (peeled, versioned) to align as many data references in the loop as possible. This can not be done without being aware of the enclosing loop context.

3. Reduction is another example for a situation where considering the basic block in the context of the loop extends the range of computations that the SLP approach can capture. This is because reduction is a special case of computations that create an inter-iteration def-use dependence cycle, and can be detected only when looking at the computation at the loop level.

4. While the loop-based approach uses the loop as the starting point for the vectorization analysis, without the loop-level context the SLP approach needs a different source to start the vectorization analysis from, and that starting point is the adjacent memory references. Therefore, if data is accessed in a strided manner with gaps (e.g., accessing only the even elements of an array), we have non-adjacent accesses that are entirely missed by the SLP approach. However, they can be easily detected as a group of accesses using our loop-based strided accesses analysis.

At the same time, loop-aware SLP improves upon the classic loop-based approach in the following ways:

1. It extends the range of computations that can be vectorized, while reusing the same infrastructure for the analysis and transformation. Examples for loops that can be vectorized were mentioned in the previous section.

2. It improves the efficiency of the generated code in cases that knowledge on how data is used within an iteration can help generate less data reorganization operations, either by eliminating them all together (as shown in the example above), or by postponing them and scheduling these operations more wisely.

3. It offers a more register-efficient vectorization approach, that can be used when the register pressure incurred by combining `VF` iterations is too high. If parallelism can be exploited by combining less than `VF` iterations together, potentially combining statements only from a single iteration, the register pressure can be significantly reduced. For example, Figure 2d uses two vector registers (`vt1`, `vt2`) to vectorize the `a`-accesses, while Figure 2c uses only one vector (`vt`) to vectorize the `a`-accesses.

The key point is that performing these optimizations requires extending our current strided accesses support to look into the uses of these strided accesses, and in that sense to continue to extend our analysis scheme in the direction of SLP. Section 4 describes how we extended the loop-based one-statement-at-a-time vectorization approach in this direction.

3 Vectorizer Overview

In this section, we describe the relevant vectorization infrastructure, to provide the necessary background for the next section where we explain how this infrastructure is extended to support loop-aware SLP.

The vectorization pass, which serves as the basis for incorporating our support for loop-aware SLP, occurs during the tree-SSA optimization stage of GCC [14]. It follows the classic loop-based vectorization approach as discussed in the previous section. It applies a set of analyses to each given loop, followed by the actual vector transformation (for loops that successfully complete the analysis phase).

The vectorizer focuses on innermost,[1] single basic block countable loops. (Certain multiple basic block constructs may be collapsed into straight-line code with conditional operations by an if-conversion pass prior to vectorization). It starts off by scanning all the statements in the loop to determine the *Vectorization Factor* (`VF`), which represents the number of different consecutive iterations of the loop that are combined into a single vector iteration in the vectorized loop (a conceptual unrolling factor). The `VF` is determined according to the data types that appear in the loop and the *Vector Size* (`VS`) supported by the target platforms. As explained below, considering SLP in a hybrid framework together with the loop-based vectorization introduces additional unrolling factors to choose from.

Next, the vectorizer finds all the memory references in the loop and checks if it can construct an access function that describes the series of addresses that each memory reference accesses across iterations of the loop (using scalar evolution analysis [18]). The access function is needed for memory-dependence, access-pattern and alignment analyses. The following memory dependence analysis pass checks that the dependence distance between every pair of data references in the loop is either zero (i.e., intra-loop dependence) or at least `VF`, by applying a set of classic data dependence tests [4], [7]. This is the point where interleaving groups are recognized and built [16].

The loop-aware SLP analysis that we describe in the next section is based on the analysis of interleaved data, i.e., data that is accessed in the loop with a non-consecutive (strided) pattern. During the dependence resolution traversal over pairs of load/store statements, groups of interleaved loads or stores that have the same stride and adjacent base address are constructed. For example, the accesses to array `a` in Figure 2a form an interleaving group of size 2 (both have a stride 2, and their bases, `a` and `a+1`, are adjacent). An interleaving group represents a consecutive range in memory, that is either fully accessed in each iteration (as in the example) or accessed with gaps (which would be the case if only the `a[2*i]` locations in the example were accessed). These groups of memory references are considered together when transforming the loop, in order to avoid redundant loads/stores and exploit data reuse.

Following the memory dependence tests and detection of interleaved data, we examine the access patterns of each memory reference and check that they access consecutively increasing addresses, or that they are a part of an interleaving group (which is the context in which we handle non-consecutive (strided) accesses).

The final analysis phase verifies that every operation in the loop can be supported in vector form by the target architecture. After the analyses are done, the loop transformation phase scans all the statements of the loop from the top down (vectorizing definitions before their uses) and inserts a vector statement in the loop for each scalar statement that needs to be vectorized. Vectorization of interleaving groups proceeds as follows: when the first member (load/store) of an interleaving group is encountered (during the top down traversal), we generate vector load/store statements starting from the lowest address accessed within the group. We then generate a complete set of $\delta * log2\delta$ data reordering statements of the form extract even/odd (for loads) and interleave low/high (for stores), where δ is the size of the interleaving group. See for example Figure 2c. Finally, the loop bound is transformed to reflect the new number of iterations, and if necessary, an epilogue scalar loop is created to handle cases of loop bounds that do not divide by `VF`. (This epilogue must also be generated in cases where the loop bound is not known at compile time.)

[1] The vectorizer is being extended to also consider some forms of doubly nested loops.

(a) Scalar:

```
for (int i = 0; i < n; i++) {
  a0 = in[4i+1];
  a1 = in[4i+2];
  a2 = in[4i];
  a3 = in[4i+3];

  b0 = X + a0;
  b1 = Y + a1;
  b2 = Z + a2;
  b3 = W + a3;

  c0 = b1 * 5;
  c1 = b3 * 6;
  c2 = b2 * 7;
  c3 = b0 * 8;

  out[4i] = c0;
  out[4i+1] = c1;
  out[4i+2] = c2;
  out[4i+3] = c3;
}
```

(b) After SLP (Pure, Full):

```
vec_inv1 = {X, Y, Z, W};
vec_inv2 = {5, 6, 7, 8};

for (int i = 0; i < len; i++) {
  v1 = vload in[4i:4i+3];
  v2 = vperm (v1, {1,2,0,3});
  v3 = vec_inv1 + v2;
  v4 = v3 * vec_inv2;
  vstore out[4i:4i+3] = v4;
}
```

Figure 3: Basic SLP example

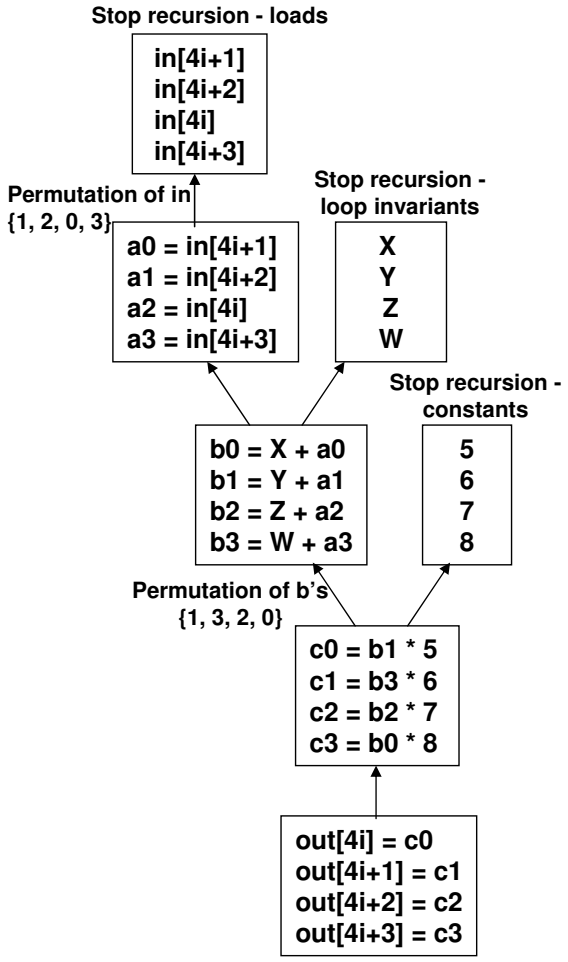

Figure 4: Computation tree for basic example

4 Extending the Vectorizer to Perform Loop-Based SLP

4.1 Analysis

We use Figure 3 to explain and demonstrate the analysis performed by our SLP.

The analysis for SLP starts by examining the results of the loop-based interleaving analysis. Referring to Figure 3(a), this interleaving analysis builds one group consisting of the first four load statements, and another group consisting of the last four store statements. Each such group contains interleaved accesses to memory. The analysis for SLP builds computation trees[2] in a recursive bottom-up, use-def manner rooted at interleaved store groups. Each node in a tree contains the same number of isomorphic statements, which can execute in parallel (i.e., can be converted into an SIMD statement). At each step, the uses of statements in the current node are examined for their corresponding defs,[3] and we either extend the tree by additional child nodes, conclude that the current node is a leaf, or terminate the tree. When the def statements are all loop-invariant (including constants) or are all loads from interleaved memory addresses (i.e., form an interleaved load group), the node is considered a leaf. Additional child nodes are created when the def statements are all loop-variant, isomorphic and independent non-load statements. In all other cases, including reaching defs which belong to SSA ϕ-nodes, we terminate the tree and delete it. Referring to the example in Figure 3(a), the SLP analysis will build a tree depicted in Figure 4 comprising of seven nodes rooted at the interleaved

[2]Not to be confused with GCC's tree structures.

[3]The SSA framework fully supports this use-def traversal.

store group, with three leaves being the interleaved load group and two loop-invariant groups (of {X,Y,Z,W} and {5,6,7,8}). Note that if a computation includes common sub-expressions (i.e., is a DAG), the analysis step will duplicate the common nodes and build a tree; however, the transformation step will recognize such redundancies and generate code only once for such duplicates.

Some computations are not captured by a bottom-up process rooted at interleaved store groups, because they do not include such stores; one prominent case being reduction computations that compute a scalar. Reductions are currently recognized and vectorized by the loop vectorizer. In order to recognize and SLP-vectorize reduction computations, one can apply a similar bottom-up process but rooted at reductions instead of interleaved store groups, or apply the opposite top-down approach starting from interleaved load groups. We chose to implement the former approach, as it fits well with the bottom-up approach originating from interleaved store groups and also with the existing loop-based framework for recognizing reductions.

The loop-vectorizer is currently capable of handling interleaving load and store groups whose size is a power-of-2, because of the data rearrangement operations needed to loop-vectorize non-unit strided accesses [16]. On the other hand, SLP is not restricted to power-of-2 sized groups: it seeks to pack statements together according to the *Vector Size* (VS)—the number of elements in a vector of the relevant type, possibly using unrolling or using several vectors for each group. Furthermore, even if the *Group Size* (GS, number of statements in group) is not a multiple of the vector size or vice versa (that is when $\gcd(GS,VS) < \min(GS,VS)$), it is still possible to unroll the loop (by $\mathrm{lcm}(GS,VS)/GS$) or pack as much as possible into vector-size groups leaving the remaining elements in scalar form. Our initial framework is restricted to cases where $\gcd(GS,VS) = \min(GS,VS)$, and marks each computation tree with the required unrolling factor (VS/GS) or the number of multiple vectors needed (GS/VS), which are integers.

Data-dependence testing for SLP is required to make sure we can move all statements of each group into one location where they are to be replaced by an SIMD statement. However, as we transform the entire computation tree together (see Subsection 4.4), these dependence tests are mostly exercised by the above use-def traversal of building the tree. Two cases require special attention: when there are multiple leaves corresponding to (interleaved) load groups, and when nodes are shared among two or more trees. In the former case, we make sure the addresses for all relevant loads are available at the location of the first load (this is already checked by the interleaving analysis of the loop-vectorizer). In the latter case, the first tree to be transformed generates the code for the mutual node, and subsequent trees (generated afterwards) will reuse it. Note that if unrolling is desired, we require inter-iteration data-dependence tests similar to those of (and already provided by, in our framework) the traditional loop-vectorizer.

4.2 SLP Taxonomy

We find it convenient to distinguish between the following two pairs of mutually complementing definitions. The first applies to the type of parallelism extracted when creating vector statements, applying SLP to a tree, or the loop in general:

Pure SLP If only intra-iteration parallelism is extracted when creating vector statements. That is, if only SLP (with GS=VS) is applied. Such cases are applicable for general computation trees within basic blocks, not necessarily within loops. In the loop context, this means that the loop is vectorized without being unrolled, using only SLP.

Hybrid SLP If both intra- and inter-iteration parallelism are extracted when creating vector statements. That is, when SLP is applied with $GS < VS$. In the loop context, this means that the loop is vectorized together with unrolling, using SLP (and possibly loop-vectorization).

The following, orthogonal terms relate to the vectorization of a loop:

Full loop vectorization If all statements in a loop are replaced by vector statements, as a result of either SLP or loop-vectorization. That is, the resulting loop contains only vector statements.

Partial loop vectorization If some statements in a loop remain in their original scalar form.

For example, the loop in Figure 3(b) is Fully vectorized using Pure SLP, the loop in Figure 5(b) is Partially vectorized using Pure SLP, and the loop in Figure 5(c) is Fully vectorized using Hybrid SLP.

(a) scalar:

```
for(i = 0; i < n; i++) {
  b[6i+0] = b0;
  b[6i+1] = b1;
  b[6i+2] = b2;
  b[6i+3] = b3;
  b[6i+4] = b4;
  b[6i+5] = b5;
}
```

(b) partial SLP, no unroll:

```
vb0 = {b0,b1,b2,b3};
for (i = 0; i < n; i++) {
  b[6i+0:3] = vb0;
  b[6*i+4] = b4;
  b[6*i+5] = b5;
}
```

(c) Full SLP, unroll=2:

```
vb0 = {b0,b1,b2,b3};
vb1 = {b4,b5,b0,b1};
vb2 = {b2,b3,b4,b5};
for (i = 0; i < n; i+=2) {
 b[12*i+0:3] = vb0;
 b[12*i+4:7] = vb1;
 b[12*i+8:11] = vb2;
}
```

(d) loop-based, unroll=4:

```
vb0 = {b0,b1,b2,b3};
vb1 = {b4,b5,b0,b1};
vb2 = {b2,b3,b4,b5};
for (i = 0; i < n; i+=4) {
 b[24*i+0:3] = vb0;
 b[24*i+4:7] = vb1;
 b[24*i+8:11] = vb2;
 b[24*i+12:15] = vb0;
 b[24*i+16:19] = vb1;
 b[24*i+20:23] = vb2;
}
```

Figure 5: Vectorization alternatives

4.3 Decision

After completing the analysis which builds computation tree candidates for SLP, we reach the decision point of how to vectorize the loop. We need to decide whether to apply SLP vectorization for each such computation tree, and also decide whether to apply loop-vectorization for remaining statements: those that do not belong to any SLP tree and those that belong to SLP trees which we decide not to SLP. The resulting loop unrolling factor is the least-common multiple of the SLP trees unrolling factors and the loop-vectorization unrolling factor.

One important factor to check when deciding to SLP a tree is alignment—if it is not loop-invariant, the alignment handling overhead is to be incurred every iteration, which is often too expensive. This happens when the stride of memory accesses (including unrolling if applied) does not divide by the vector size. When the alignment is loop-invariant, loop-versioning and peeling are used to extract this overhead from within the loop. In our initial approach we handle interleaved accesses without gaps, so the stride is equal to GS, and we restrict our attention to cases where GS divides VS (possibly after unrolling), so the alignment is indeed loop-invariant. Partial loop vectorization, where only a subset of statements from each node in a computation tree is SLP'ed and the rest remain in scalar form, may encounter loop-variant alignment problems.

Another factor to consider is the number of iterations, bearing in mind the unrolling factors.

The order of elements within the groups of a tree may not be consistent throughout the computation, thereby requiring the insertion of element-permuting statements. Our bottom-up approach is reminiscent of the *eager shift* [26] policy of alignment permutations, where data is to be reordered as early as possible according to the order desired by the final store group. Common subexpressions in the loop may imply nodes that are shared among several trees (or shared several times within a tree)—they can be directly reused if all uses require the same order of elements within the group, otherwise an overhead of permuting the elements needs to be considered. In some cases, statements of a computation tree cannot be loop-vectorized (for example, due to non power-of-2 strides); therefore, in Full vectorization mode we decide to perform SLP in such cases.

We have yet to devise a cost-model mechanism that integrates these considerations; our current plans are to gather statistics in order to better understand the trade-offs among these various factors.

4.4 Transformation

After deciding which computation trees to SLP and whether to loop-vectorize the remaining statements or not, we perform the actual code transformation by a top-down scan of the loop's original scalar statements in their original order. Upon reaching the first statement of a computation tree to be SLP'ed (typically the first load), we generate the vectorized version of the

whole tree, from leaves to root; if any node of the tree has already been generated by an earlier (or the same) tree, we reuse it. Otherwise, if we decided upon loop-vectorization the statement is vectorized in the traditional way, and if not we leave the scalar statement intact. The result of applying SLP to the loop in Figure 3(a) is given in Figure 3(b).

5 Future Work

We extended the GCC vectorizer to support SLP. An initial implementation is available in the autovect-branch of GCC. There are many directions in which the current framework can be enhanced. We review most of them in this section.

1. Support MIMD (Multiple Instructions Multiple Data) parallelism, such as the "subadd" computation illustrated in the example in Figure 1c, as well as other forms of mixtures of non-isomorphic defs that could still be combined into a vector. There are targets that can directly support certain MIMD operations (SSE3, for example, supports a subadd vector operation), but it can be vectorized also on targets that don't directly support it (by multiplying the relevant vector operand by a vector of 1s and –1s).

2. As explained in Section 4 the vectorizer considers different unrolling factors as it attempts to maximize the exploitation of SIMD parallelism. Currently this is limited to cases in which the `GS` (the *Group Size*) is less than `VS` (the *Vector Size*) and evenly divides `VS`. This restriction can be relaxed to also consider unrolling the loop in cases where `GS` divides $VF * \alpha$, for some integer α. This is illustrated in Figure 5c, in which `GS` is 6, the `VS` is 4, and unrolling the loop by a factor of 2 maximizes the exploitation of SIMD parallelism.

3. The current implementation does not address the case in which the `GS` is greater than VS and not all the elements of the group are defined by isomorphic computations, but there exists a subgroup of `VS` elements that are defined by isomorphic computations. The current implementation attempts to construct the SLP-tree from the entire group, and will therefore fail and terminate. However, the analysis can continue if the implementation is extended to explore subgroups of size `VF` of the SLP-group under consideration.

4. The current implementation terminates when it reaches a ϕ-node. Therefore, computations that involve a vectorizable cross-iteration dependency cannot be vectorized. Such computations include reductions, inductions, and also false-dependencies formed by optimizations like PRE and predictive commoning. The SLP-analysis can be extended to analyze beyond ϕ-nodes.

5. The current implementation does not support partial vectorization, however the framework is general enough to consider vectorizing only part of the loop and leaving the rest of the statements unvectorized, as shown in Figure 5b, where 4 statements are grouped into a vector statement, and 2 statements remain unvectorized.

6. As already mentioned in Section 4, by examining the uses of interleaved stores, a loop-based vectorizer could postpone the rearrangement of data to a later stage (this is demonstrated in more detail in [16]). It is free to decide when to rearrange the data—either immediately when loading (resembling the eager shift heuristic in [26]), or at a later stage of the computation not originally associated with loads or stores (resembling the lazy shift heuristic). Ultimately, we may want to reorder in anticipation of the permutation needed by the stores, if internal operations are isomorphic, similar to the eager shift scheme. A simple case to optimize occurs when the rearrangement at the stores is the inverse of that at the loads (e.g., `interleave low/high` and `extract odd/even`), thereby canceling each other (this raises the issue of combining and optimizing permutations [19]).

7. By starting the SLP analysis off of the pre-analyzed interleaving groups, the current framework easily supports SLP vectorization when there are strided accesses with gaps. This is, however, not yet included in the current implementation.

8. SLP can be used to vectorize within an iteration in situations where there are cross-iteration dependencies that prevent loop-based implementation. The current implementation applies the SLP analysis however is applied only after full dependence testing has passed. This restriction should be relaxed.

6 Related Work

SLP was first introduced by Larsen and Amarasinghe [9], exploiting SIMD parallelism in basic blocks. Their algorithm starts from identifying seeds of statements with adjacent memory references, packs them into group and then merges the groups to get groups of the size of SIMD instruction. Later Larsen *et al.* [11] presented methods for forcing congruence among the dynamic addresses of memory references, and thus increasing the number of memory operations that can be grouped into SIMD instructions. Shin *et al.* [21] extended the concept of SLP so that it can be applied in the presence of control flow constructs by using if-conversions. Compiler-controlled caching in superword register files, an SLP-complementary optimization, was shown by Shin *et al.* [20]. The implementations were done within SUIF compiler infrastructure using AltiVec instruction set. Our approach for SLP differs in its use of our adjacent memory references analysis, originally developed for vectorizing loops with strided accesses, and in having implemented SLP within the loop-based framework taking advantage of loop-level analyses. The approaches mentioned above usually compensate for the lack of loop context by carefully optimizing alignment and picking appropriate unrolling factors in advance.

Larsen *et al.* [10] introduced techniques for selective vectorization of instructions. Our approach also allows partial loop vectorization and generating loops with a mixture of scalar and vector operations.

Proprietary compilers like Intel's icc and IBM's xlc have also incorporated SLP techniques into their vectorization framework. Wu *et al.* [25] proposed a simdization framework in xlc that extracts SIMD parallelism from different program scopes, including basic blocks and inner loops, based on an intermediate representation of virtual vectors. The Intel compiler [6] applies an early SLP pass that upon detecting statements that can be replaced by SIMD instructions converts them into explicit loops (by re-rolling the code) which are later vectorized using the regular loop-based vectorization pass.

Several studies focus on vectorizing computations that involve data permutation. Kudriavtsev and Kogge [8] followed the SLP approach for contiguous data, using an abstract source-to-source framework, for data permutation optimization. Ren, Wu, and Padua [19] optimized sequences of permutations for IBM's VMX and Intel's SSE2. Our framework allows supporting data permutations, though this feature has not been added yet.

Tenllado *et al.* [23] built a modified SLP compiler for Intel SSE to efficiently exploit vector parallelism from the outer loop in loop nests that process 2D arrays. By incorporating transposition (data permutation) into the SLP packing stage, and scheduling several unroll-and-jam and regular unrolling passes at different nesting levels, they are able to exploit parallelism from external nests (i.e., across different rows). GCC currently does not have an unroll-and-jam optimization, and our vectorization is thus currently limited to packing statements from at most doubly-nested loops.

Some architectures provide SIMOMD (Single Instruction Multiple Operations Multiple Data) [5, 22]. The Vienna MAP source-to-source vectorizer [12] is applicable to such architectures and was shown to produce most efficient code, focusing on straight-line 2-way vectorization by exploiting domain knowledge for specific algorithms such as FFT. Our implementation currently targets only classic SIMD instructions, but the new extension that looks at def-use chains within an iteration allows us to incorporate this capability into our vectorizer as well.

7 Conclusions

The SLP approach to vectorization can exploit parallelism in important computations that cannot be vectorized otherwise. We have incorporated SLP vectorization techniques into the loop-based GCC vectorizer, thereby complementing and extending its vectorization capabilities, while re-using the same infrastructure. In this paper we described how this was done, as well as what are the tradeoffs and considerations involved. We also showed why incorporating SLP in a loop-aware context also improves upon the original SLP approach, and how it was used to create a hybrid loop-aware SLP approach that can exploit parallelism both across loop iterations as well as inside the loop. The initial implementation presented in the paper, a first step of an ongoing work, can be enhanced in several directions, that will further improve the applicability and efficiency of the GCC vectorizer.

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] Cell Broadband Engine. `http://www-306.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine`.

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

[5] 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 *In Proc. of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACTŠ04)*, pages 85–96, September 2004.

[6] A.J. Bik. *The Software Vectorization Handbook: Applying Multimedia Extensions for Maximum Performance.* Intel Press, Hillsboro, OR, 2004.

[7] G. Goff, K. Kennedy, and C.W. Tseng. Practical Dependence Testing. In *Proceedings of the SIGPLAN Conference on Programming Languages Design and Implementation*, June 1991.

[8] A. Kudriavtsev and P. Kogge. Generation of Permutations for SIMD Processors. In *Proc. of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 147–156, June 2005.

[9] 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.

[10] S. Larsen, R. Rabbah, and S. Amarasinghe. Exploiting Vector Parallelism in Software Pipelined Loops. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 119–129, 2005.

[11] S. Larsen, E. Witchel, and S. Amarasinghe. Increasing and Detecting Memory Address Congruence. In *Proc. of International Conference on Parallel Architectures and Compilation Techniques*, pages 18–29, Sep 2002.

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

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

[14] D. Novillo. Tree SSA - A New Optimization Infrastructure for GCC. In *Proc. of the GCC Developers Summit*, pages 181–193, May 2003.

[15] D. Nuzman and R. Henderson. Multi-platform Auto-vectorization. In *Proc. of the 4th Annual International Symposium on Code Generation and Optimization (CGO)*, pages 281–294, March 2006.

[16] 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)*, pages 132–143, June 2006.

[17] D. Nuzman and A. Zaks. Autovectorization in GCC – two years later. In *Proc. of the GCC Developers Summit*, pages 145–158, June 2006.

[18] S. Pop, A. Cohen, and G.A. Silber. Induction Variable Analysis with Delayed Abstractions. HiPEAC Journal, November 2006.

[19] G. Ren, P. Wu, and D. Padua. Optimizing Data Permutations for SIMD Devices. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 118–131, June 2006.

[20] 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.

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

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

[23] C. Tenllado, L. Pinuel, M. Prieto, F. Tirado, and F. Catthoor. Improving superword level parallelism support in modern compilers. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 303–308, September 2005.

[24] M. Wolfe. *High Performance Compilers for Parallel Computing*. AddisonWesley, 1996.

[25] P. Wu, A.E. Eichenberger, A. Wang, and P. Zhao. An Integrated Simdization Framework Using Virtual Vectors. In *Proceedings of the 19th Annual International Conference on Supercomputing*, pages 169–178, June 2005.

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