Improving GCC's auto-vectorization with if-conversion and loop flattening for AMD's Bulldozer processors

Sebastian Pop Reza Yazdani Quentin Neill
Open Source Compiler Engineering, Advanced Micro Devices, Inc.
sebastian.pop@amd.com, reza.yazdani@amd.com, quentin.neill@amd.com

Abstract

The automatic vectorization of GCC relies on the if-conversion pass to remove control flow from the innermost loops. Another transformation that removes control flow in loop nests is loop flattening: it extends the applicability of if-conversion to loop nests, collapsing nested loops into a single loop. Loop flattening creates new opportunities for optimizations such as vectorization and VLIW software pipelining which are designed for loops with no nesting. Loop flattening together with if-conversion can transform non perfectly nested loops into a form that can be vectorized using predicated vector code.

The upcoming AMD's Bulldozer processors introduce a conditional move instruction, part of the XOP instruction set, that enables the efficient execution of predicated vector code. To exploit this capability, we improved the if-conversion and we added a new loop flattening pass to GCC. We present preliminary results of the improved vectorization capabilities of GCC on AMD Bulldozer processors.

1 Introduction

The work that we describe in this paper examines ways to enable GCC's automatic vectorization of loops with irregular control flow. Currently the vectorizer only handles loops that are under a canonical form: a unique basic block contains all the code of the loop body and the loop latch basic block is empty. When one of these characteristics is not satisfied in a loop, the vectorization is not applied.

The if-conversion transformation of GCC can transform a simple loop with conditions into the canonical form expected by the vectorizer. When looking at the control flow graph, the if-conversion removes forward edges and leaves the code of the loop body in a single basic block. We describe the restrictions under which the if-conversion cannot be applied and the kind of loops that can be vectorized after if-conversion. We then propose a technique that enables if-conversion on more cases by avoiding the restrictions.

In order to extend GCC's automatic vectorization to arbitrarily complex loop nests, we explored loop flattening transformations in two GCC representational domains: Graphite and Loop-Cfg-Gimple-SSA. Loop flattening collapses several loops either nested or that are executed in sequence into a single loop. Once flattened, any conditions in the resulting loop can be if-converted in order to obtain the canonical form expected by the vectorizer.

When looking at the effect of the loop flattening on the control flow graph, backward pointing edges are transformed into forward edges. This remark resulted in a general algorithm that removes all back-edges of a loop body, collapsing them into its own latch edge. In fact, this general algorithm applies as well to unstructured code that currently cannot be handled by the natural loop analysis.

In the next sections, we describe the if-conversion transformation and its implications on vectorization. Then we explore two loop-flattening algorithms, both the high level semantical algorithm that we implemented using the Graphite infrastructure and the low level back-edge removal algorithm that we implemented in GCC on the Loop-CFG-Gimple-SSA representations.

2 If-conversion

In the absence of side-effects, if-conversion transforms control-flow into data-flow (i.e., control dependences are transformed into data dependences [1]) with the use of conditional expressions. This transformation removes edges pointing forward in the CFG improving the

capabilities of the automatic vectorization pass of GCC. Other positive effects of if-conversion are:

- branch mis-prediction: as there are fewer branches after if-conversion, there are fewer mis-predicted branches;
- instruction pipeline: the loop's steady state is more regular on the if-converted form and allows better throughput avoiding bubbles in the pipeline;
- scheduler: because of a larger basic block, the scheduler is less restricted to reorder the execution of instructions;
- ILP: the amount of instruction level parallelism in the loop is increased by if-conversion as several independent execution paths are merged together.

As a simple example of if-conversion, the following loop containing a condition:

```
for (i = 0; i < N; i++) {
  if (cond)
    a = x;
  else
    a = y;
  b = a + 2;
}
```

is transformed into this loop that has no control flow in its body:

```
for (i = 0; i < N; i++) {
  a = cond ? x : y;
  b = a + 2;
}
```

When the transformation is performed on a Static Single Assignment (SSA) form, as implemented in GCC, the phi nodes at the junction of two incoming edges are replaced with a conditional expression. On the previous example written in SSA form:

```
loop
  if (cond)
    a_0 = x;
  else
    a_1 = y;
  a_2 = phi (a_0, a_1);
  b_3 = a_2 + 2;
end_loop
```

if-conversion removes the phi node and replaces it with a conditional expression as follows:

```
loop
  a_0 = x;
  a_1 = y;
  a_2 = cond ? a_0 : a_1;
  b_3 = a_2 + 2;
end_loop
```

2.1 Restrictions for if-conversion

In the original program, only one of the branches of the condition is taken and evaluated for a given iteration of the loop. After if-conversion, both branches of the condition are evaluated. This transformation is not always legal, as the non taken branch might contain side effects that the original program did not have. The usual example of illegal if-conversion contains a NULL pointer check and a dereference of the pointer:

```
for (i = 0; i < N; i++) {
  if (p)
    *p = 3;
}
```

that might produce an exception, if the condition is if-converted, due to the dereference of a NULL pointer.

If-conversion might increase the number of memory accesses, potentially introducing data races in multi-threaded code. The typical form of illegal if-conversion are due to the unconditional execution of statements that could trap[1], for example:

- writes in a read only memory,
- accesses of out-of-range memory,
- accesses of invalid pointers,
- division by zero.

2.2 If-conversion of memory reads and stores

If we assume a memory model for a single threaded cpu[2], it is possible to also if-convert memory stores under the additional conditions:

- the data references used in the if-converted branches occur either in read or write mode at every iteration of the loop, guaranteeing that the data references will not access out-of-range memory more than the original program,
- there exists at least one unconditional write to the array, guaranteeing that an array is not const qualified.

[1] See Joseph S. Myers's remarks on the legality of if-conversion: `http://gcc.gnu.org/ml/gcc-patches/2010-06/msg02362.html`

[2] For instance -fmemory-model=single as discussed in `http://gcc.gnu.org/ml/gcc/2010-05/msg00171.html`

Under these conditions, the if-conversion that we implemented in GCC would transform this code

```
for (i = 0; i < N; i++)
  if (cond)
    A[i] = expr;
```

by introducing a full write to A[i] at every iteration:

```
for (i = 0; i < N; i++)
  A[i] = cond ? expr : A[i];
```

2.3 Results

We have extracted several kernels from the FFmpeg source code that have been vectorized by hand for MMX or SSE2 together with the C version of the kernel. A very common pattern in these kernels is the use of conditionals in a loop. We analyzed more precisely one of these kernels used in the encoding and decoding of video streams:

```
#include <stdlib.h>

#define N 256 * 256

static void
dct_unquantize_h263_inter_c (int * block,
  int qscale, int nCoeffs)
{
  int i, level, qmul, qadd;

  qadd = (qscale - 1) | 1;
  qmul = qscale << 1;

  for (i = 0; i <= nCoeffs; i++)
    if (block[i] < 0)
      block[i] = block[i] * qmul - qadd;
    else
      block[i] = block[i] * qmul + qadd;
}

int
main (void)
{
  int *block = (int *) malloc (N * sizeof (int));

  for (i = 0; i < N; i++)
    block[i] = i % 2;

  dct_unquantize_h263_inter_c (block, 42, N);

  return block[10];
}
```

After if-conversion and code optimizations the DCT loop kernel looks like this, and is vectorized:

```
  for (i = 0; i <= nCoeffs; i++) {
    t1 = block[i];
    t = t1 < 0 ? - qadd : qadd;
    block[i] = t1 * qmul + t;
  }
```

We generated two versions, one targeting the SSE2 vector instructions and another version targeting XOP. We used a simulator for the upcoming Bulldozer processors to evaluate the effectiveness of the vectorization using the XOP instructions VPCMOV (conditional move) and VPMAC (multiply accumulate) with respect to the SSE2 instruction set.

The XOP version executed 2.51 times faster than the SSE2 version. Part of that is due to the fact that the XOP version executes fewer instructions: the XOP version executes 163882 instructions and the SSE2 version that executes 540714 instructions. The SSE2 emulates the predicated code with a combination of logical operations whereas the VPCMOV executes as a single instruction. Furthermore, the XOP version uses the fused multiply accumulate VPMAC instruction instead of two successive operations.

2.4 If-conversion without restrictions

In order to avoid the above mentioned restrictions, and systematically enable if-conversion, the compiler can generate code that is valid and free of trapping operations by introducing a pointer indirection. The compiler can create an object that is written to or read from in the case of the branch that is not taken: a pointer is used to redirect the output to the useful object that the original program uses or to the artificial object that has been created by the compiler.

For the undefined pointer example, the compiler could generate the following if-converted code, with q a pointer to a valid object created by the compiler:

```
for (i = 0; i < N; i++) {
  t = p ? p : q;
  *t = 3;
}
```

This same technique could be used to avoid excessive write accesses to the memory used by the original program, making the if-conversion safe for multi-threaded programs. For instance, on the following code:

```
for (i = 0; i < N; i++) {
  if (cond)
    A[i] = expr;
}
```

instead of if-converting by introducing a full write to A[i] at every iteration, the compiler could generate this other version that would be thread-safe, with q defined,

as before, as a pointer to a valid object created by the compiler:

```
for (i = 0; i < N; i++) {
  t = cond ? &A[i] : q;
  *t = expr;
}
```

This same technique could apply to the other restriction cases previously mentioned: writes to a read-only memory, out-of-range accesses, and access of invalid pointers.

For the case of division by zero, we should assume in Gimple that the conditional operations exists for all operations including the divide. It is up to the RTL code generation to generate an if statement, if the operation does not exist in the hardware.

3 Loop flattening on a high level semantical representation of loops

Loop flattening [2] transforms loop nests into a single loop, removing the loop nesting structure. The auto-vectorization can then apply on the full loop body, without needing the outer-loop vectorization.

The canonical example is as follows: suppose that we have a loop nest with known iteration counts

```
for (i = 1; i <= 6; i++)
  for (j = 1; j <= 6; j++)
    S1(i,j);
```

The loop flattening is performed by linearizing the iteration space using the function $x = 6 * i + j$. In this case, GCC with Graphite would produce this code:

```
for (x=7;x<=42;x++) {
  i = floord(x-1,6);
  S1(i,x-6*i);
}
```

The limitations of this loop flattening implementation are linked to the limits of the polyhedral model: the linearizing function should be affine and this excludes the multiplication of parameters with iteration variables. To allow the transformation of such cases, one has to take an upper bound approximation of the parameter. For example, in the following loop nest,

```
for (i = 1; i <= N; i++)
  for (j = 1; j <= M; j++)
    S1(i,j);
```

one would like to flatten this loop using the linearizing function $x = M * i + j$. However linear schedules are not expressive enough to deal with this case, and so the parameter M has to be replaced by an integer upper bound approximation. If we further know that $M \leq 6$, then it is possible to linearize the loop with $x = 6 * i + j$. In this case, Graphite would produce this code:

```
for (x=7;x<=6*M+N;x++) {
  i = ceild(x-N,6);
  if (i <= floord(x-1,6)) {
    S1(i,x-6*i);
  }
}
```

For an arbitrarily complex loop nest the algorithm proceeds in two steps. First, the Loop Statement Tree (LST) is flattened by removing the loops structure and by inserting the statements in the order they appear in depth-first order. Then, the scattering of each statement is transformed accordingly.

Supposing that the original program is represented by the following LST:

```
(loop_1
 stmt_1
 (loop_2 stmt_3
  (loop_3 stmt_4)
  (loop_4 stmt_5 stmt_6)
  stmt_7
 )
 stmt_2
)
```

Loop flattening traverses the LST in depth-first order, and flattens pairs of nested loops successively by projecting the inner loops in the iteration domain of the outer loops. In the flattened loop, the execution of the statements of the outer loop are interleaved with the execution of the statements of the inner loop. In this example, we suppose that the number of iterations of the loops is represented by $stride_l$ indexed by the loop number l. The projection starts from the innermost loops towards outer loops: $loop_3$ is projected into $loop_2$, then $loop_4$ is projected into $loop_3$ and finally $loop_2$ is projected into $loop_1$ as follows:

```
lst_project_loop (loop_2, loop_3, stride_3)

(loop_1
 stmt_1
 (loop_2 stmt_3 stmt_4
  (loop_4 stmt_5 stmt_6)
  stmt_7
 )
 stmt_2
)

lst_project_loop (loop_2, loop_4, stride_4)
```

```
(loop_1
 stmt_1
 (loop_2 stmt_3 stmt_4 stmt_5 stmt_6 stmt_7)
 stmt_2
)

lst_project_loop (loop_1, loop_2, stride_2)

(loop_1
 stmt_1 stmt_3 stmt_4 stmt_5 stmt_6 stmt_7 stmt_2
)
```

At each step, the iteration domain of the outer loop is enlarged to contain enough points to iterate over the inner loop domain. When flattening a nested loop into an outer loop, the number of iterations of the resulting loop is the product of the number of iterations of the inner loop by the number of iterations of the outer loop. When flattening two sequential loops, the number of iterations of the resulting loop is the sum of the number of iterations of both loops.

3.1 Legality of loop flattening

Loop flattening might look similar to loop fusion as the flattening of two sequential loops produces a single loop. However the way the two loops are merged differs: loop fusion interleaves the iterations of the first loop with the iterations of the second loop leading to a loop nest that has the same number of iterations, whereas loop flattening juxtaposes the iterations of the second loop after the execution of all the iterations of the first loop. Loop fusion has to check the legality of the transformation as the interleaving of iterations can cause data dependence violations. Loop flattening is a transformation that is always legal as it does not change the execution order of the original loops.

3.2 Vectorization

At this moment, the code generated by Graphite would not be vectorizable, as the old induction variables are now containing modulo and division expressions of the new single linearized induction variable. The scalar evolution framework does not handle such expressions and thus the new data reference accesses cannot be analyzed. David Edelsohn proposed to perform the auto-vectorization on the polyhedral representation where the memory accesses are still represented as linear functions in a higher degree vector space.

This is the main limitation of using a high level sementical representation of loops in Graphite to perform loop flattening when the goal is automatic vectorization of the final loop.

4 Loop flattening by elimination of back-edges

A generalization of the loop flattening does not consider the loop semantics as presented in the previous section, and can apply to unstructured code in which the backward pointing edges of the control flow graph (CFG) have been identified.

4.1 Overview

The back-edge removal transformation was described in [1] as part of the if-conversion algorithm for backward pointing edges. In this section we will first provide a description of this technique adapted for the Gimple-SSA form, followed by an example, and a discussion of the differences with the higher level loop flattening transformation.

The back-edge removal algorithm transforms control dependences into data dependences by using a boolean variable. The values taken by the boolean variable control the execution path of the forward edges created in order to use the back-edge of an outer loop.

The first step of the algorithm detects a surrounding loop and all the back-edges of the loop body: these back-edges can be inner loops or strongly connected components of the CFG that cannot be reduced to natural loops.

Each back-edge is removed by redirecting the target of the back-edge to the latch basic block of the surrounding loop (this basic block will be named “latch” in the rest of this section). A boolean variable is created in the latch. It is cleared when the redirected back-edge is taken and it is set to true for any other paths leading to the latch.

The header basic block of the surrounding loop (this basic block will be named “header” in the rest of this section) is split before its statements and a new condition is added based on the control variable: when the control variable is set to true, the execution proceeds as normal to the basic block that contains the statements of the header; when the control variable is cleared, meaning that the back-edge has been taken, the execution proceeds to the point where the redirected back-edge was pointing.

The last step updates the SSA form after all the back-edges have been redirected to the latch, and the new edges from the header to the destination of back-edges have been created.

4.2 Example

The following example illustrates the back-edge removal:

```
int foo (void)
{
  int i, j;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      A[i][j] = i + j;
  return A[12][3];
}
```

after loop flattening the code contains a single loop:

```
int bar (void)
{
  int i, j, flag = 1;
  for (i = 0; i < N;)
    {
      if (flag)
        j = 0, flag = 0;

      if (j < N)
        {
          A[i][j] = i + j;
          j++;
        }
      else
        i++, flag = 1;
    }
  return A[12][3];
}
```

and after if-conversion, the code would look like this:

```
int bar (void)
{
  int i, j, t;
  int *q = &t;
  bool flag = true;
  bool cond;

  for (i = 0; I < N; i = !cond ? i + 1 : i)
    {
      // flag is true on entry to the loop
      j = flag ? 0 : j;
      flag = false;
      cond = j < N;
      p = &A[i][j];
      p = cond ? p : q;

      // A[i][j] written only if cond is true
      *p = i + j;
      j = cond ? j + 1 : j;
      flag = !cond ? true : flag;
    }
}
```

4.3 Implications

The back-edge removal transformation is a more general form of loop flattening which applies to a wider set of code forms. In contrast to the loop flattening in the Graphite framework, it is not restricted to natural loops and not restricted to the analysis of the number of iterations and data access functions. However, the code produced by the back-edge removal has different characteristics than the code produced by loop flattening: boolean flags are used and there are more forward edges produced.

This general loop flattening approach is useful before software pipelining or scheduling in machines with multiple processing elements. In particular if the number of iterations of inner loop is small, the outer loop can benefit from one basic block scheduling and software pipelining.

4.4 Vectorization

The back-edge removal transformation is harmful for the induction variable analysis: the induction variables are not incremented systematically at each iteration of the resulting loop, and thus they are not simple enough to be handled by the scalar evolution analysis. This disables all the loop transformations on this loop nest, and so the vectorization would not happen on such code.

5 Conclusion

In SIMD machines or heterogeneous execution models such as OpenCL where the same instruction is executed with different data, branches are usually performance bottlenecks, because they may take different destinations in different units. Removal of the branch from the code makes it a more efficient execution model.

We presented several techniques that we implemented and contributed to GCC to improve the vectorization of loop kernels containing control flow. We presented some preliminary results of if-conversion with vectorization and a more general if-conversion technique to remove the restrictions of if-conversion for memory reads and stores.

Finally the paper presented two different approaches to loop flattening that could expose more instruction level parallelism and enable more vectorization.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 177–189, New York, NY, USA, 1983. ACM.

[2] R. von Hanxleden, R. Hanxleden, K. Kennedy, and K. Kennedy. Relaxing simd control flow constraints using loop transformations. In *In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 188–199, 1992.