Reprinted from the

Proceedings of the GCC Developers' Summit

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

Profile driven loop transformations

Richard Günther
SUSE Labs
rguenther@suse.de

Abstract

Today scientific computing applications are developed using modern principles of software design. Among others, this leaves specialization and optimization of loop kernels to the compiler. In particular, loops which run for a known low number of iterations in one of the dimensions, such as loops handling boundary condition computation, usually produce inferior code using F95 array expressions or C++ template library utility functions. The same holds true for strides of multidimensional arrays which usually are the same for all arrays that participate in a loop kernel, but still are not known so at compile time.

GCC has developed a rich infrastructure for both loop analysis and transformation[1] as well as supporting profile guided optimizations[3]. Using this infrastructure we present the results of developing loop optimizations that rely on the use of loop versioning and profiling of iteration counts and access evolution. The goal is to reduce the numbers of induction variables to consider during induction variable optimization and improve the generated code by requiring a less overall number of registers. This is done by providing loop specializations for both the above mentioned cases. We present the implementation of the instrumentation and the optimizing phase discussing current limitations of the framework GCC provides. A case study involving the TraMP3d benchmark to gather statistical data for the transformations is presented as well as performance results for applying the transformations on a standard loop kernel.

1 Introduction

There are two different species of profiles, *CFG profiles*, which profile for instance edge execution counts, and *value profiles*, which profile for instance the value of the dividend in a division instruction.

A CFG profile can be used to direct inlining decisions such as emphasizing inlining into hot sections of a program and keep cold sections optimized for size. It also is used to estimate branch probabilities to guide partitioning of hot and cold code sections and basic block reordering to improve code locality and instruction cache efficiency. A CFG profile can to some extent also be used to estimate loop iteration counts to guide optimizations such as loop unrolling and peeling, though in general CFG profile info is too imprecise here.

Value profiles on the other hand are used to decide whether specializations of computations are worthwhile, such as specializing a division instruction for a constant divisor or dividend. In general value profile using optimizations increase code size for introducing a new common faster path based on knowledge of certain values used in the following computation.

A simple example for a value profile transformation is the instruction

```
c = a/b
```

which can be transformed into

```
if (b == 1) c = a;
else c = a/b;
```

based on profile information that `b == 1` is indeed common.

In the following, we will use value profiles of variables and predicates composed from quantities used in nested loop code to create specializations of these loops which can then be optimized by later passes.

1.1 Interesting Loops

We are interested in a certain type of loops. This is not necessarily a requirement of the instrumentations and transformations we outline, but we will restrict further discussion on nested loops of the form

```
for (iN = i0N; iN < i1N; ++iN)
 ...
  for (i0 = i00; i0 < i10; ++i0)
    f (mem0[(i0 + C00) * stride00
            + ... + (iN + CN0) * strideN0],
       ...,
       memM[(i0 + C0M) * stride0M
            + ... + (iN + CNM) * strideNM]);
```

That is, loops of the nest N which have an inner loop body that is a function of M memory reads or writes with possibly different access patterns specified by the strides stride00 to strideNM and the constant offsets C00 to CNM. Interesting loops are required to have loop invariant strides, loop bounds and memory base addresses.

GCC has infrastructure to analyze and identify loops of the above form and canonicalize them so that the above constraints can be verified by looking at the SSA web. In particular, the scalar evolution infrastructure[1] can be used to verify loop invariantness and to query the number of iterations of each loop of the nest. It also provides a way to decompose memory access patterns to the form outlined above.

1.2 Transformations

The set of transformations we are after is inspired by the TraMP3d[2] hydrodynamics code. TraMP3d is based on the FreePOOMA[4] library which provides data-parallel array operations similar to Fortran90+. As those go through a common loop expander template function, common degenerate cases are worth to optimize. The loop expander template for three dimensions looks like the following

```
template <class LHS, class Op, class RHS,
          class Domain>
inline static void __attribute__((flatten))
evaluate(const LHS& lhs, const Op& op,
         const RHS& rhs, const Domain& domain)
{
  int e0 = domain[0].length();
  int e1 = domain[1].length();
  int e2 = domain[2].length();
#pragma omp parallel for
  for (int i2=0; i2<e2; ++i2)
    for (int i1=0; i1<e1; ++i1)
      for (int i0=0; i0<e0; ++i0)
        op(lhs(i0,i1,i2), rhs.read(i0,i1,i2));
}
```

Here memory access patterns such as strides and constant offsets are encapsulated in the `lhs` and `rhs` expression template[5] objects.

Let `op` do a simple assignment of the `rhs` object to the `lhs` object. Further be `lhs.strides` an array specifying the strides used to access the `lhs` memory, and `rhs.strides` for the `rhs` memory. `domain.size` should be an array specifying the size of the domain to iterate over. We are then interested in the following specializations being done:

(a) lhs.strides[0] == rhs.strides[0] == 1

(b) lhs.strides[0] == rhs.strides[0] == 1 and domain.size[0] == 2

(c) lhs.strides[0] == rhs.strides[0] == 1 and domain.size[1] == 2

(d) lhs.strides[0] == rhs.strides[0] == 1 and domain.size[2] == 2

Here (a) covers copying of unit stride array regions, and (b) to (d) cover periodic boundary updates. In addition to these specializations, reflecting boundary condition updates would be optimized by versioning for lhs.strides[0] == 1 and rhs.strides[0] == -1 and the respective domain size conditions.

By providing specializations for these cases, we rely on the following optimization passes to take advantage about the additional knowledge.

- Induction variable optimization is presented with a problem reduced in complexity due to the now partially constant strides.
- The linear loop transformation pass can decide to move the low-iteration count loop either to the outermost or the innermost nest, which allows
- either loop unrolling to unroll the innermost loop completely,
- or induction variable optimization to consider the most expensive updates for the outermost loop to improve induction variable selection for the inner nests.

2 Preparation

To be able to use the infrastructure mentioned above we need to first do some cleanup transformations on the loops, namely loop header copying, which transforms the loops into do-while loops, and loop invariant motion by using load-PRE to make loop invariant memory references regular SSA variables in the SSA web of the loop nest. To achieve this, we have moved the tree profiling pass to a later point in the optimization pipeline and inserted a set of optimization passes before it. The relevant part of the optimization pipeline now looks like (inserted passes marked with *):

Initial scalar cleanups:

```
  pass-ccp
  ...
  pass-dce
```

Kill empty loops getting in the way of loop analyzing. The WrapNoInit template leaves us with empty loops counting from 2 to -1 otherwise.

```
  pass-tree-loop-init
* pass-empty-loop
* pass-complete-unroll
* pass-tree-loop-done
```

The VRP pass above exposed new forwprop opportunities (which in turn exposes copyprop opportunities) due to folding casts again. And another may-alias pass to expose the store copyprop opportunities to DOM.

```
* pass-forwprop
* pass-may-alias
  pass-dominator
  ...
  pass-ch
```

We need a load-PRE pass to hoist loads of loop invariant strides and counts out of the loop bodies. Possible due to loop header copying.

```
* pass-split-crit-edges
* pass-pre
* pass-may-alias
* pass-hoist-guards
```

We need a copyprop pass to have the same SSA names for loop tests as the hoisted loads from PRE.

```
* pass-rename-ssa-copies
* pass-copy-prop
* pass-dce
* pass-tree-loop-init
```

Try getting rid of extra PHIs inserted by loop-init.

```
* pass-phi-only-cprop
  pass-tree-profile
```

The early loop pass is to get rid of unrelated inner loops in the TraMP3d benchmark, likewise the extra forward propagation and may-alias passes are to expose SFTs of array elements to the following DOM pass doing store copy propagation.

Entering the tree-profile pass, a properly optimized loop looks like that in Fig. 1.

From the optimization pipeline you can see that we needed to make the tree profiling code work on SSA form. This was necessary anyway because the infrastructure for loop modification and the SCEV analysis requires SSA form. In the current state profiling is now done after inlining, so profile-based inlining is disabled. After the merge of the IPA-branch it will be possible to place the early optimizations and the profiling before the final inlining pass, which is then done on SSA form.

2.1 Limitations of the current infrastructure

The current profiling and loop infrastructure presents us with several limitations that have been partially addressed for this work. First of all, the existing profile instrumentation pass does not work on SSA form, while SCEV analysis and all optimization passes require that. This has been fixed and allows moving the `tree-profile` pass to a later point in the optimization pipeline.

SCEV analysis for figuring out the number of iterations of a loop needs to be improved to deal with more cases that happen for example in TraMP3d. The situation with this has been improved by us and Sebastian Pop. For example we were not able to determine the number of iterations of the loop for (int i=i0; i<=i1+1; ++i);, which runs i1-i0+2 times if it runs at all.

Loop header copying, SCEV and load-PRE interact in interesting ways, in particular with loop nests. We and Zdenek Dvorak have developed several ideas to work around these issues for the testcases we looked at sofar. Still there are cases where invariant loads are not hoisted out of the loops even if they are known to iterate at least once. This sometimes makes analyzing of memory accesses impossible, see Fig. 2 for an illustration of the difficulty to hoist stride loads.

```
int e2 = d.sizes[2];
int e1 = d.sizes[1];
int e0 = d.sizes[0];
int s2 = a.stride[2];
int s1 = a.stride[1];
int s0 = a.stride[0];
if (e2 > 0)
  if (e1 > 0)
    if (e0 > 0)
      {
        int k=0;
        do {
          int j=0;
          do {
            int i=0;
            do {
              a.m[i*s0+j*s1+k*s2] = 0.0;
              ++i;
            } while (i<e0);
            ++j;
          } (while j<e1);
          ++k;
        } (while k<e2);
      }
```

Figure 1: Properly optimized loop-structure for analyzing with SCEV. All invariant memory loads have been hoisted out of the loops.

```
for (int k=0; k<d.sizes[2]; ++k)

  for (int j=0; j<d.sizes[1]; ++j)

    for (int i=0; i<d.sizes[0]; ++i)
      ... *stride;
```

```
if (d.sizes[2] > 0)
  do { k=0;
    if (d.sizes[1] > 0)
      do { j=0;
        if (d.sizes[0] > 0)
          do { i=0;
            ... *stride;
          ++i; } while (i<d.sizes[0]);
      ++j; } while (j<d.sizes[1]);
  ++k; } while (k<d.sizes[2]);
```

Figure 2: Loop header copying interaction with PRE. All loop header copies need to be hoisted out of the outermost loop for PRE to hoist the load of *stride out of the outermost loop. Compare to properly optimized loop structure in Fig. 1

Further, the current CFG profile instrumentation code does not preserve loop structure information, as it inserts new basic blocks due to instrumenting edges. Also the infrastructure for loop versioning only deals with non-nested loops. We didn't fix any of those problems yet, but restricted the transformations done to produce one loop version, which then can afford to destroy loop information.

For optimization of the versioned loop the biggest problem at the moment is the missing ability of the value range propagation pass to deal with a series of predicates of the form `A && B`. This happens if we version for example the loop in Fig. 3 for the stride condition `D.3300289_111`. VRP in this case only handles nested conditional statements, not a single conditional statement with a composed condition.

In future work also need to teach the linear loop transformation pass to consider exchanging loops not only due to data locality reasons but also considering the possibility to remove a loop nest completely due to unrolling. This is an important transformation as later measurements will show.

3 Implementation

The implementation of the loop profiling pass is divided into an analysis phase and an instrumentation or transformation phase, dependent on the mode of compilation. The analysis is done before the CFG profiling analysis and instrumentation, while the instrumentation is done after CFG profiling has cleaned up after itself. Profile counters are read and stored alongside the analysis data.

3.1 Loop analysis

During analysis we walk the loop tree and search for loop nests with a depth of at least two where each of the loop satisfies the following constraints:

- The loop has a single exit edge.
- The loop has at most one direct child.
- The number of iterations can be symbolically computed at compile time and is invariant in the whole interesting nest.

The last constraint ensures we can insert instrumentation code on the exit edge of the outer loop and that we can use the symbolic number of iterations in the transformation phase for the loop versioning condition. We rely on PRE to move the necessary defining statements to before the loop header copies.

For each such nest found, we walk the innermost loop list of basic blocks to identify and analyze memory loads and stores for their access strides using the scalar evolution of the memory address. To be interesting for profiling, those strides need to be invariant with respect to the outermost loop. From this data we compute a single boolean value that is true if all innermost loop strides are one. Building this value in a different way, or computing multiple values for other conditions is easily possible, too. A useful extension would be to check if all memory access strides for the innermost loop are the same, or to handle loads and stores separately for the checks. Another useful property to profile is alignment and data dependency, information that can be used for example by the vectorizer.

3.2 Loop instrumentation

In the instrumentation phase we insert single-value profiling counters for the conditions built during the memory access stride analysis. The outcome of the profile is then if the condition was true or false most or all of the time.

For the loop iteration counts we insert interval profiling counters with an interesting range of one to two, which gives us exact counts for once and twice iterating loops as well as the overall number of times the nest was entered.

All profiling counters are inserted on the outermost loop single exit edge, so the instrumentation is cheap and all counts are relative to the number of invocations of the whole loop nest. Instrumenting on the exit edge requires all instrumented values definition site to dominate the insertion place, which restricts the set of possibly instrumented values to loop invariant ones. The instrumentation place also requires previous optimization passes to hoist all these invariants out of the loop nest, which is possible due to canonicalization to do-while loops done by loop header copying. In Fig. 3 the instrumented code after the tree-profiling pass is shown for a selected loop from the TraMP3d-4 benchmark.

```
<bb 2>:
  D.3300088_50 = lhs->domain_m[0].domain_m[1];
  D.3300097_65 = lhs->domain_m[1].domain_m[1];
  D.3300106_80 = lhs->domain_m[2].domain_m[1];
  if (D.3300106_80 > 0) goto <L34>; else goto <L12>;
<L34>:;
  if (D.3300097_65 > 0) goto <L42>; else goto <L12>;
<L42>:;
  if (D.3300088_50 > 0) goto <L47>; else goto <L12>;
<L47>:;
  D.3300117_133 = rhs->data_m;
  D.3300119_143 = rhs->strides_m[0];
  D.3300121_150 = rhs->strides_m[1];
  D.3300124_82 = rhs->strides_m[2];
  D.3300144_177 = lhs->data_m;
  D.3300146_90 = lhs->strides_m[0];
  D.3300148_2 = lhs->strides_m[1];
  D.3300151_168 = lhs->strides_m[2];
  # i2_146 = PHI <i2_94(13), 0(5)>;
<L30>:;
  D.3300125_3 = D.3300124_82 * i2_146;
  D.3300152_155 = i2_146 * D.3300151_168;
  # i1_139 = PHI <i1_99(11), 0(6)>;
<L24>:;  D.3300283_151 = (long long int)
  D.3300122_5 = i1_139 * D.3300121_150;
  D.3300149_67 = D.3300148_2 * i1_139;
  # i0_6 = PHI <i0_194(9), 0(7)>;
<L5>:;
  D.3300120_126 = i0_6 * D.3300119_143;
  D.3300123_129 = D.3300122_5 + D.3300120_126;
  D.3300127_132 = D.3300125_3 + D.3300123_129;
  D.3300128_134 = (unsigned int) D.3300127_132;
  D.3300129_135 = D.3300128_134 * 8;
  D.3300130_136 = (double *) D.3300129_135;
  D.3300131_137 = D.3300117_133 + D.3300130_136;
  D.3300133_138 = *D.3300131_137;
  D.3300147_170 = i0_6 * D.3300146_90;
  D.3300150_173 = D.3300149_67 + D.3300147_170;
  D.3300154_176 = D.3300152_155 + D.3300150_173;
  D.3300155_178 = (unsigned int) D.3300154_176;
  D.3300156_179 = D.3300155_178 * 8;
  D.3300157_180 = (double *) D.3300156_179;
```

```
  D.3300159_181 = D.3300144_177 + D.3300157_180;
  a_188 = D.3300159_181;
  *a_188 = D.3300133_138;
  i0_194 = i0_6 + 1;
  if (D.3300088_50 > i0_194) goto <L5>; else goto <L7>;
<L7>:;
  i1_99 = i1_139 + 1;
  if (D.3300097_65 > i1_99) goto <L24>; else goto <L9>;
<L9>:;
  i2_94 = i2_146 + 1;
  if (D.3300106_80 > i2_94) goto <L30>; else goto <L55>;
<L55>:;
  D.3300280_140 = (unsigned int) D.3300106_80;
  D.3300281_114 = (long long int) D.3300280_140;
  __gcov_interval_profiler (&*.LPBX2[0], D.3300281_114, 1, 2);
  D.3300282_104 = (unsigned int) D.3300097_65;
  D.3300282_104;
  __gcov_interval_profiler (&*.LPBX2[4], D.3300283_151, 1, 2);
  D.3300284_193 = (unsigned int) D.3300088_50;
  D.3300285_187 = (long long int) D.3300284_193;
  __gcov_interval_profiler (&*.LPBX2[8], D.3300285_187, 1, 2);
  D.3300287_92 = D.3300146_90 == 1;
  D.3300288_109 = D.3300119_143 == 1;
  D.3300289_111 = D.3300287_92 && D.3300288_109;
  D.3300290_161 = (long long int) D.3300289_111;
  __gcov_one_value_profiler (&*.LPBX4[0], D.3300290_161);
<L12>:;
  return;
```

Figure 3: Instrumented loop from the TraMP3d-v4 benchmark (arc profiling code removed).

3.3 Loop transformation

The transformation phase decides whether specializations for low iteration count or constant one inner strides are worthwhile. The latter is done in case the strides proved to be one all the time, while the former is decided based upon the fraction of the times the count was low compared to the overall loop nest invocations. A value of $1/7$ has been extracted from the TraMP3d profile counts, which enables the wanted transformations in the boundary update routine. Unfortunately, if loop versioning using a condition combined from sub-conditions using logical AND, later VRP passes are not able to extract value ranges from this compound conditional, so the effect of the transformation phase is limited until this is fixed.

4 Application statistics

We have run the loop instrumentation on the TraMP3d-v4 benchmark application and collected profile data to verify if we can successfully identify worthwhile transformations. The profile collecting run invoked 10 iterations of the benchmark, which ensures that all loop nests are entered at least once. In TraMP3d-v4 there are 63 interesting loops with different kernels produced by the generic expander function templates. Out of these, for 8 loops we can successfully instrument memory access strides and 6 of these have all inner strides equal to one for all invocations. For all 63 loops we can insert counters for the number of iterations of the loops, 8 of them happen to be never executed because of benchmark flag settings.

The remaining not instrumented strides are due to our inability to load-PRE the memory accesses to the strides out of the loop nest, which causes SCEV analysis to punt on the memory accesses. Improvements in this area are subject of future work.

In the profile using compilation we schedule the periodic boundary update kernel for three specializations, one for each dimension with its iteration count being two and the innermost loop strides equal to one. The profile data in this case tells us that each of the low-iteration specialization will run 315 times, while the unversioned copy will run 2151 times. A total of 945 times, 30% of the loop nest invocations, will be spent in optimized versions of the loop.

All of the 6 loops that have inner strides equal to one for all invocations get an extra optimized loop version created.

5 Performance experiments

Performance experiments have been carried out using a small benchmark application that mimics the basic structure of an abstract C++ array operations framework like it is used in TraMP3d. The loop kernel used for the experiments is a simple assign operation applied using the following loop:

```
template <class Op>
void expand (const Array& a1, const Array& a2,
             const Domain& d, const Op& op)
{
  int ie = d.sizes[0];
  int je = d.sizes[1];
  int ke = d.sizes[2];
  for (int k=0; k<ke; ++k)
    for (int j=0; j<je; ++j)
      for (int i=0; i<ie; ++i)
        op(a1(i,j,k), a2.read(i,j,k));
}
```

with `op` assigning the second argument to the first for the benchmark and the array accesses implemented as `storage[i*strides[0] + j*strides[1] + k*strides[2]]`.

The first experiment was to compare performance of the loops with the transformations outlined above applied to the unmodified loop. Specifically, comparing the original loop (`normal`) with the loops with inner stride optimization (`stride`), unrolling and loop exchange (`iter`) and the loops with both optimizations applied (`both`). The results are as follows.

i686	normal	stride	iter	both
full	1411	1371	1462	1471
dim0	390	333	164	159
dim1	137	125	189	98
dim2	109	101	189	97
amd64				
full	1581	1493	1606	1531
dim0	237	201	158	170
dim1	128	112	159	63
dim2	114	100	148	63
ppc				
full	2394	2380	2355	2351
dim0	785	803	141	144
dim1	317	314	141	145
dim2	284	284	142	146

The numbers are milliseconds for 100000 invocations of the loop kernel operating on 64 kB of memory for full accesses and 8 kB for the partial accesses. Thus, memory bandwidth should be that of the L1/L2 cache. The first column specifies the type of data access, `full` being a complete copy of one array to the other, while `dim0` to `dim2` copy slices of width two in dimension N (0 being the innermost loop), simulating boundary updates.

One can clearly see that for the simple loop kernel reducing the number of induction variables is a win only for register-starved machines such as the i686, while on ppc the most gain in performance is with the low iteration count loop unrolled in the innermost nest position.

With profiling enabled we are able to obtain the same performance results with training runs restricted to one of the partial accesses. We can also see that it would generate all the `both` specializations for `dim0` to `dim2` if training with the full set of tests.

Due to the limitations outlined above we were unable to produce meaningful performance numbers for the TraMP3d benchmark. Work on the FreePOOMA library has shown though, that loop kernel specialization for innermost strides being one are very important for at least register starved machines. Similar experiments with optimization for low iteration count have been unsuccessful due to the explosion in number of template function instantiations caused by a non-profile feedback directed approach. Here, the profile based optimization is the solution once the remaining issues are solved.

6 Conclusions

We have shown that with some work, the loop and profiling infrastructure of GCC can be used to perform loop specializations crucial for heavy templated C++ code. In particular, relying on the compiler to identify optimization opportunities for boundary update loops makes it possible to no longer manually specialize those routines. Compared to tackling the same problem with template specialization, the profiling approach leads to much lower compile time and text size of the resulting program.

We also have identified certain weak spots in the infrastructure of GCC. There is work ongoing to improve GCC in this regards and we hope to provide the community with a cleaned up infrastructure for profiling and loop optimization in the 4.3 time frame.

Ultimately the goal should be to generally make CFG manipulation loop aware and manage loop structure as a required or provided pass property. This should also allow to propagate information gathered from the profiles to later passes such as the loop unroller and the vectorizer.

References

[1] David Edelsohn Daniel Berlin and Sebastian Pop. High-level loop optimizations for gcc. In *The 2004 GCC Developers' Summit Proceedings*, June 2004.

[2] Richard Günther. *Three-dimensional Parallel Hydrodynamics and Astrophysical Applications*. PhD thesis, Eberhard-Karls-Universität Tübingen, 2005.

[3] Jan Hubicka. Profile driven optimizations in gcc. In *GCC Developers' Summit Proceedings*, June 2003.

[4] John V. W. Reynders, Paul J. Hinker, Julian C. Cummings, Susan R. Atlas, Subhankar Banerjee, William F. Humphrey, Steve R. Karmesin, Katarzyna Keahey, Marikani Srikant, and Mary Dell Tholburn. POOMA: A Framework for Scientific Simulations of Paralllel Architectures. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming in C++*, pages 547–588. MIT Press, 1996.

[5] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995.