Using MELT to improve or explore your GCC-compiled source code

Basile STARYNKEVITCH
basile@starynkevitch.net (or basile.starynkevitch@cea.fr)
CEA, LIST (Software Reliability Lab)
CEA/DRT NanoInnov bat962 PC174 91191 GIF/YVETTE CEDEX, France

April 16, 2012

Abstract

This paper introduces the MELT framework and domain-specific language to extend the GCC compiler. It explains the major internal representations (*Gimple*, *Tree*-s, ...) and the overall organization of GCC. It shows the major features of MELT and illustrates why extending and customizing the GCC compiler using MELT is useful (for instance, to use GPGPUs thru OPENCL). It gives some concrete advices and guidelines for development of such extensions with MELT.

Keywords: compilation, GCC, domain-specific language, High-Performance Computing, GPGPU, OpenCL, MELT, *Gimple*.

1 introduction to MELT

1.1 GCC and MELT

MELT [8, 9] (see `http://gcc-melt.org/` for code and documentation) is a high-level domain specific language[1] to extend or customize the GCC compiler. GCC (see `http://gcc.gnu.org` for more) is the major free software compiler for many languages (C, C++, Ada, Fortran, Go, Objective C . . .), target processors (x86, ARM, Sparc, MIPS, PowerPC, . . .) and systems (Linux, AIX, Android, . . .).

The GCC compiler is a large legacy software (with more than 5 millions source lines [2], which provides (notably in its 4.6 release of march 2011, and its 4.7 release of march 2012) a plugin machinery to extend it. But coding plugins for GCC in plain *C* is quite painful (because *C* is not a language specially suited for symbolic processing, like the one happening inside compilers). MELT is a *domain specific language* designed to ease the development of specific GCC extensions, for tasks aiming specific applications or libraries like: specialized warnings, specific optimizations, coding rules validation etc.

A GCC compilation reads the user source code, then transforms it in specific GCC internal representations, notably *Tree*-s (for declarations and operands) and *Gimple*-s (for instructions). The bulk of an optimizing GCC compilation is made of more than 250 GCC passes operating on its internal representations. GCC uses many internal representations, notably *Gimple*-s ("normalized" elementary abstract instructions, representing elementary steps of the compiled code, such as the addition of a temporary "variable" with a constant going into another temporary) and *Tree*-s (elementary abstract syntax trees, notably operands of *Gimple*-s, e.g. temprary, local or global variables, formals and constants).

MELT can be used to take advantage of GCC powerful representations and processing by customizing it suitably to the (advanced) user's needs. It is implemented as a "meta"-plugin for GCC, which generates *C* code operating on GCC internal representations, then compiles that generated *C* into MELT modules and dynamically loads these modules. MELT can be executed on GNU/LINUX systems[3] having a GCC compiler supporting plugins.

Working inside GCC enables accesss to the internal representations of the compiler. This has several benefits: it is more precise than textual based tools (for instance, it is quite easy to find all functions with two formal double arguments without requiring their definition's signature to fit on a single line); since it is working on some cuirrent internal representations in the compiler, it can profit of prior work already done by the compiler (parsing -for various source languages-, previous optimization and inlining

[1]The MELT implementation is free software (GPLv3+ licensed) available from `http://gcc-melt.org/` as a plugin for GCC (4.6, 4.7 or better).

[2]Measuring the source code size of GCC is already a challenge. Some tools may give nearly 10 millions lines of source code lines as the computed code size of the same version of GCC.

[3]MELT should be easily ported to any operating system providing the POSIX `dlopen` and `dlsym` functions for loading dynamic shared libraries, and supporting GCC plugins.

```c
void matmul(int n, double*m, double *a, double *b)
{
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++) {
      double s=0.0;
      for (int k=0; k<n; k++)
        s += a[i*n + k]  *  b[k*n + j];
      m[i*n + j] = s;
}
```

Figure 1: matrix multiplication `matmul.c`

passes) and could improve its future work (remaining optimization or code generation passes).

1.2 A glimpse into *Gimple*

Let's consider the classical square matrix multiplication code in *C*, with each matrix represented by an array of double precision IEE754 numbers. The *C* source code is given by figure 1 (and is an obvious candidate to use the GPU, for large enough matrix size `n`). From the compiler's point of view, this code might not be parallelizable, e.g. when `m` and `a` matrixes are physically overlapping in memory, and we should declare the formals as `double* restrict m` etc to tell the compiler that they cannot overlap or alias.

This simple code is transformed a lot inside the GCC compiler. Very quickly, it gets "gimplified", that is transformed into *Gimple* instructions. To obtain a textual dump of internal representations after most passes of GCC, invoke it as `gcc-4.7 -std=c99 -O2 -Wall -fverbose-asm -fdump-tree-all -c matmul.c`; this gives a lot of (arbitrarily[4] numbered) dump files, including `matmul.c.004t.gimple` figure 2, which shows that the *Gimple* form contains much more elementary instructions, and many temporary variables like e.g. `D.1603`; the compiler introduces these to break complex instructions into simpler operations. Control flow operations are broken up into many jump labels e.g. `<D.1591>` and elementary tests with `goto`-s.

After "gimplification", many other GCC optimization passes transform that high-level *Gimple* representation into lower-level *Gimple/SSA* (for *Static Single Assignment*). This is a form in which every variable is assigned once (so can be understood as defined once with a sort-of equality, so is "simpler" to handle thru formal methods). For instance, our matrix multiplication is later optimized into the dump file `matmul.c.086t.phiopt2` of 91 lines, including figure 3, which illustrates that the *Gimple/SSA* form uses several SSA versions (e.g. `j_33` and `j_44`) of the same Gimple variable (here `j`). When a basic block (like `<bb 7>` of figure 3) is reachable from several defining assignments (in different blocks) for a variable, a Φ [function] assign-

[4]The numbers in dump file names are unique, but sadly don't order these dump files in chronological executions of passes.

```
matmul (int n,
        double * m, double * a, double * b)
{
  int D.1595;
  int D.1596;
  long unsigned int D.1597;
  long unsigned int D.1598;
  double * D.1599;
  double D.1600;
  int D.1601;
  int D.1602;
  long unsigned int D.1603;
  long unsigned int D.1604;
  double * D.1605;
  double D.1606;
  double D.1607;
  int D.1608;
  long unsigned int D.1609;
  long unsigned int D.1610;
  double * D.1611;
  {   int i;
    i = 0;
    goto <D.1592>;
    <D.1591>:
    {   int j;
      j = 0;
      goto <D.1589>;
      <D.1588>:
      {      double s;
        s = 0.0;
        {        int k;
          k = 0;
          goto <D.1586>;
          <D.1585>:
          D.1595 = i * n;
          D.1596 = D.1595 + k;
          D.1597 = (long unsigned int) D.1596;
          D.1598 = D.1597 * 8;
          D.1599 = a + D.1598;
          D.1600 = *D.1599;
          D.1601 = k * n;
          D.1602 = D.1601 + j;
          D.1603 = (long unsigned int) D.1602;
          D.1604 = D.1603 * 8;
          D.1605 = b + D.1604;
          D.1606 = *D.1605;
          D.1607 = D.1600 * D.1606;
          s = D.1607 + s;
          k = k + 1;
          <D.1586>:
          if (k < n) goto <D.1585>;
          else goto <D.1587>;
          <D.1587>:
        }
        D.1595 = i * n;
        D.1608 = D.1595 + j;
        D.1609 = (long unsigned int) D.1608;
        D.1610 = D.1609 * 8;
        D.1611 = m + D.1610;
        *D.1611 = s;
      }
      j = j + 1;
      <D.1589>:
      if (j < n) goto <D.1588>;
      else goto <D.1590>;
      <D.1590>:
    }
    i = i + 1;
    <D.1592>:
    if (i < n) goto <D.1591>;
    else goto <D.1593>;
    <D.1593>:
  }
}
```

Figure 2: gimplification of matrix multiplication `matmul.c.004t.gimple`

ment statement explicits that fact by "merging" the several origins of that variable's value, so `i_41 = PHI<i_34(6), 0(2)` means that the versioned SSA name i_{41} gets its value either from i_{34} computed in `<bb 6>` or from `0` when coming from `<bb 2>`.

Notice that coding the same algorithm in the different programming languages understood by GCC produces very similar *Gimple/SSA* forms inside the GCC compiler, even when using high-level *C++11* constructs like nested `std::vector` or `std::valarray` templates, `std::for_each` application, and `<functional>` standard header with anonymous functions; the standard *C++11* library is nicely optimized by GCC, but that library is still implemented thru usual *C++11* constructs.

1.3 MELT features

MELT is a high-level domain specific language designed to ease the extension of GCC for customization purposes. It is implemented in a free-licensed (GPLv3) GCC plugin (and MELT infrastructure), has the following features:

- The MELT language has a Lisp-like syntax and design : every MELT phrase is an expression producing some result[s]. Every expression (called S-expr) is in parenthesis, starting with the operator or keyword, and followed by operands.

```
<bb 3>:
  # s_51 = PHI <s_25(3), 0.0(5)>
  # k_52 = PHI <k_26(3), 0(5)>
  D.1595_10 = i_41 * n_6(D);
  D.1596_11 = D.1595_10 + k_52;
  D.1597_12 = (long unsigned int) D.1596_11;
  D.1598_13 = D.1597_12 * 8;
  D.1599_15 = a_14(D) + D.1598_13;
  D.1600_16 = *D.1599_15;
  D.1601_17 = k_52 * n_6(D);
  D.1602_18 = D.1601_17 + j_44;
  D.1603_19 = (long unsigned int) D.1602_18;
  D.1604_20 = D.1603_19 * 8;
  D.1605_22 = b_21(D) + D.1604_20;
  D.1606_23 = *D.1605_22;
  D.1607_24 = D.1600_16 * D.1606_23;
  s_25 = D.1607_24 + s_51;
  k_26 = k_52 + 1;
  if (n_6(D) > k_26)
    goto <bb 3>;
  else
    goto <bb 4>;

<bb 4>:
  # s_40 = PHI <s_25(3)>
  D.1595_27 = i_41 * n_6(D);
  D.1608_28 = D.1595_27 + j_44;
  D.1609_29 = (long unsigned int) D.1608_28;
  D.1610_30 = D.1609_29 * 8;
  D.1611_32 = m_31(D) + D.1610_30;
  *D.1611_32 = s_40;
  j_33 = j_44 + 1;
  if (n_6(D) > j_33)
    goto <bb 5>;
  else
    goto <bb 6>;


<bb 5>:
  # j_44 = PHI <j_33(4), 0(7)>
  goto <bb 3>;

<bb 6>:
  i_34 = i_41 + 1;
  if (n_6(D) > i_34)
    goto <bb 7>;
  else
    goto <bb 8>;

<bb 7>:
  # i_41 = PHI <i_34(6), 0(2)>
  goto <bb 5>;
```

Figure 3: excerpt of `matmul.c.086t.phiopt2` - *Gimple/SSA* form

Case in identifiers (called *symbols*, as in Lisp or Scheme) is not significant. Identifiers may contain non-letter characters. So the sum of 2 and `x` is expressed as `(+i x 2)` where `+i` is a symbol denoting the addition (of two raw integers) primitive. Parenthesis are highly significant; every non-trivial MELT expression is parenthesized, except some traditional syntactic sugar [5].

- MELT is translated to *C* code (thru a translator itself implemented in MELT); that generated *C* code[6] may then be compiled into a shared object on the fly, then dynamically loaded with `dlopen`, all during the same single execution of GCC augmented by MELT. This enables MELT code to contain some small *C code chunks* which can call external libraries, or low-level GCC internal functions.

- MELT deals with "high-level" dynamically typed *values* (objects, lists, closures, tuples, boxed numbers, boxed *Gimple*-s, etc...) and raw GCC *stuff* (e.g. raw `long`, raw *Gimple*, raw *Tree*-s, raw basic blocks), and enable reflective[6] and functional programming styles.

- a runtime (implemented as the `melt.so` GCC plugin) provides many utilities, notably an efficient MELT generational copying garbage collector (tuned for frequent allocation of many temporary MELT values), built above the GCC mark

[5]The MELT reader parses `'1` exactly as `(quote 1)`, and `?(cstring_same "fflush")` exactly as `(question (cstring_same "fflush"))`, etc.

[6]The GCC compiler (version 4.7) is itself in transition, aiming to be progressively and partly rewritten in *C++* code. MELT actually is translated to the common subset of *C* and *C++* code which is acceptable inside GCC plugins. If and when GCC would be re-written in idiomatic *C++* code, the MELT translator would be adapted to that.

and sweep garbage collector (which is quite primitive, and expects most *Gcc* stuff to live quite a long time), and a dynamic loader for MELT modules.

- the MELT system has an extensible *pattern matching framework*: extensions written in MELT use many patterns and can easily filter the internal representions of GCC (notably *Gimple* and *Tree*-s) thru a high-level declarative syntax.
- the MELT implementation is layered: the runtime is used by all the MELT system; the translator parses MELT code (from file, memory, or even sockets) into s-expressions, which are then macro-expanded, normalized, generated an internal representation of the emitted *C* code, at last pretty-printed to generated *C* files, compiled into a shared object, dynamically loaded. Each layer is quite modular and can be extended by an advanced user. For instance, user-specific language constructs can be provided (thru the macro machinery).
- the MELT system is able to deal with asynchronous textual messages[7] (in MELT S-expr syntax); this enables passes coded in MELT to communicate with external processes (e.g. graphical interfaces, web servers, ...) thru asynchronous textual protocols, in messages which might be handled at nearly any time inside passes provided in MELT (but not inside existing GCC passes, which would not react to them).

The MELT plugin release 0.9.5 (available on `http://gcc-melt.org/`, free software, GPLV3+ licensed, of april 2012) for GCC 4.7 has a MELT runtime of 23KLOC (thousands of line of source code, counted with `wc`), including nearly 6KLOC of MELT generated code, the MELT (to C or C++) translator has 46KLOC (giving 1629KLOC of generated *C* code, which is also distributed), and nearly 8KLOC of MELT code providing a foundation for user provided MELT extensions.

[7]Since version 0.9.5 of april 2012 of the MELT plugin for GCC 4.7

2 MELT usage for HPC customization

HPC (High-Performance Computing) should become a preferred domain for compiler customization, and MELT can be the right tool for that.

2.1 Traversing GCC internal representations with MELT

Working inside the GCC compiler enables accessing the exact internal representations used by the compiler. In particular, MELT permits to navigate on the code, as worked on by GCC, e.g. *after function inlining*.

To give a simple toy example (algorithm 1) [8] figure 4, one might want to catch all calls like `fflush(NULL)` (used to flush all `<stdio.h>` files) inside every function whose name starts with `bar` and do that search *after inlining*. This simple query cannot be handled by simple textual tools (like `perl` or `awk`, etc...) because it needs to know what functions are inlined by GCC (and inlining decisions are not easily predictable). The relevant internal GCC representation is *Gimple/SSA*.

```
for each function cfun do
    if cfun's name start with bar then
        for each basic block bb of cfun do
            for each gimple g inside bb do
                if g is a call to fflush with one argument, constant 0 then
                    inform the user of the location of g
                end
            end
        end
    end
end
```

Algorithm 1: finding `fflush(NULL)` inside any `bar` function

The MELT code of figure 4 illustrates several traits of the MELT programming language. Pattern matching is done with `match` expressions (beginning line 3 and 10), containing matching clauses starting with patterns (prefixed with a question mark `?`, e.g. `?_` for the wildcard pattern). Iterative constructs (beginning line 1, 6 and 8), conventionally named with `each` or `with` in their names, have input arguments [perhaps an empty list `()` if none is required] followed by local formals. So the `eachgimple_in_basicblock` expression starting line 8 can be understood as: iterate inside the raw basic block `bb`, and for each *Gimple* inside, bind it to local variable `g` whose type `:gimple`, i.e. raw *Gimple* stuff of GCC, then evaluate the body sub-expression[s] (here, a `match` on `g`) from lines 10 to 16.

This example illustrates the power of pattern matching, when analyzing (or transforming) GCC internal *Gimple* representations : filtering internal code representa-

[8]This is `ex05` of `melt-examples` on GitHub.

```
 1 (with_cfun_decl ()
     (:tree cfundecl)
 3   (match cfundecl
       ( ?(tree_function_decl_named
 5          ?(cstring_prefixed ”bar”) ?_)
          (each_bb_current_fun ()
 7            (:basic_block bb)
              (eachgimple_in_basicblock (bb)
 9              (:gimple g)
                (match  g
11               ( ?(gimple_call_1  ?_
                     ?(tree_function_decl_named
13                     ?(cstring_same ”fflush”) ?_)
                     ?(tree_integer_cst 0))
15                 (inform_at_gimple g  ”found_fflush(NULL)”))
                 ( ?_ ()))))))
17     ( ?_ ())))
```

Figure 4: MELT example - seeking `fflush(NULL)` *after inlining* inside functions `bar*`

tion is an essential operation when working inside a compiler. The `match` of line 10 of figure 4 filter a given gimple instruction `g` to find if it is a call to `fflush` with a constant `0` argument (since `NULL` is macro expanded to e.g. `(void*)0`). it is filtered as a *Gimple* call instruction with one argument (with the `gimple_call_1` matcher) to a function, represented as a *Tree* in *Gimple*, whose declaration (matching the `tree_function_decl_named` matcher) refer to a name matching the sub-pattern `?(cstring_same "flush")`, filtering strings equal to the literal `"flush"`. Actually, this is not the most efficient way to find call to the standard `fflush` function; a more clever way would be to filter all the functions declarations to find out and store the *Tree* for that `fflush` and filter for it.

Pattern matching in MELT can also *extract* components from filtered data; hence a pattern like :

```
?(gimple_assign_plus
  ?(tree_ssa_name ?var ?_ ?lvers ?_)
  ?(tree_ssa_name ?var ?_ ?rvers ?_)
  ?(tree_integer_cst ?rincr))
```

will filter all *Gimple/SSA* statements adding the *same* variable `var` (with two different *SSA* versions: `lvers` on the left side, `rvers` on the first right side of the assignment) to some integer constant `rincr`; the extracted pattern variables `var lvers rvers rincr` are available inside the body of that matching clause. The pattern is non-linear, since the same pattern variable `?var` appears twice in it. When matching its second occurrence (for the first right hand operand of the assignment), it should be compared to the already found data.

2.2 Taming HPC parallelism with GRAPHITE and MELT

An important issue in HPC is to improve parallel computation. This is a very ambitious and a very long-term goal. Within the GCC compiler, a lot of efforts have been invested for that goal, notably thru the GRAPHITE project [10, 11], which uses polyhedral techniques to detect parellelization opportunities in *Gimple*.

Using GPGPU for the highly parallel and data regular parts of the computation is possible, notably with the CUDA proprietary language and with the OPENCL industry standard [5]. Both are specialized *C*-like dialects and runtime infrastructures designed to take advantage of GPGPUs. However, they require recoding the crucial parts (i.e. numerical kernels) of an HPC application in another language (e.g. OPENCL or CUDA)

Experimental code has been developed to translate *Gimple* to *OpenCL* using the GRAPHITE infrastructure within the GRAPHITE-OPENCL project [1]. However, all GPGPU related effort within GRAPHITE is (or was) still *highly experimental*, so is *not yet integrated inside the* released GCC *compiler*. Since MELT is targeting official GCC releases[9], it is not possible to use this experimental GRAPHITE-OPENCL work from MELT. And the GRAPHITE component of released GCC compilers does not even provide any plugin hooks, so cannot be easily extended (either in MELT or thru any GCC plugin coded in *C*). Because MELT is extending `gcc-4.7` as released, it can only use the public interfaces provided by GCC.

However, the `graphite` pass of GCC 4.7 is able to detect parallelizable loops, and this detection can be used by MELT extensions. MELT knows about GCC `loop` stuff and provides a primitive `loop_can_be_parallel` to query if GRAPHITE has detected if a given loop is parallelizable. Once such a loop is found, a MELT pass has to recompute some informations that GRAPHITE has computed for its own internal purpose, without providing an API to query it. In particular, each parallelizable loop has some induction variable, but GRAPHITE is not publishing it, so MELT has to recompute it.

Hence, re-doing in MELT what has been done in *experimental prototypes* related to GCC is not easy, because the released GCC compilers does not provide the interface to do that, so a significant amount of code has to be re-written (or we need to wait until some future release of GCC provides the relevant plugin hooks or public API, or gets a newer GRAPHITE merged inside.). And merging into MELT the experimental GRAPHITE is as difficult as merging it into the GCC trunk[10] (we leave that task to the developers of GRAPHITE). Furthermore, the experimental MELT branch is very often merged with the current GCC trunk.

Hence, generating of simple OPENCL variant from the internal *Gimple* is not yet achieved (in april 2012) within the MELT project. And when tentatively running, it would only translate very simple cases of parellisable algorithms. A production-level OPENCL generator would have to be tightly integrated inside GRAPHITE.

A practical issue, when generating OPENCL, is to estimate the size of the arrays (and the number of iterations inside loops). In very simple cases (e.g. a loop with

[9]The official GCC releases are evolving significantly fast enough, so integrating experimental *Graphite* work is not reasonable for a single person outside of the small *Graphite* developers' community, and integrating some of their code into GCC releases [i.e. the current trunk] is already a challenge for *Graphite* people.

[10]The GCC trunk is the actively developed GCC branch, which will become the next release.

a compile-time constant bound), GCC may already have such an estimate. In real cases, even as simple as the trivial matrix multiplication of figure 1, determining the bounds from the *Gimple* representation of *Gimple* code from figure 2 is not that simple. Concretely, running the matrix multiplication on the GPGU is worthwhile only for a large enough (e.g. $n > 200$ perhaps) dimension of the matrix. The exact threshold -which is highly system and machine specific- should probably be computed by machine learning techniques [3].

Many other tools to take advantage of GPGPU exist. Some tools, in particular STARPU [2], extend GCC with appropriate pragmas to ease integration with "codelets" running on the GPGPU.

2.3 Why HPC needs GCC extensions and customizations?

High performance computing may profit from *specific* GCC extensions and customizations, like:

- navigation or refactoring tools for large scientific (legacy) code
- dynamically choosing an implementation of some classical numerical library (e.g. a GPU or a CPU variant of LAPACK [11], etc..)
- application specific pragmas (or compiler builtins)
- customized optimizations, in particular when mathematical knowledge may suggest them (e.g. replacing the addition of some matrix M with itself [12] by a scalar multiplication by 2)
- "domain specific profiling" could be achieved by *automatically* inserting "profiling instructions" (perhaps as simple as incrementing a profiling counter) within the *Gimple* code
- etc ...

The main insight is that some important HPC software could consider *customizing* the GCC compiler for *their own purposes*, and MELT can be the right tool for such an approach. However, extending GCC (even with MELT) has some human cost, because the GCC internal representations are intrinsically complex : *Gimple* has 38 cases (of which 14 are specific to OPENMP) listed in file `gcc/gimple.def` of GCC [see figure 5], and *Tree* has nearly 200 cases, listed in file `gcc/tree.def` [see figure 6]. A given GCC compiled application is very likely to use the majority of these cases, which have to be handled individually. Notice that some *Tree* cases cannot appear at *Gimple/SSA* level (e.g. `RETURN_EXPR`, which is rendered by a `GIMPLE_RETURN` instruction)) while others (like `SSA_NAME`) are specific to *Gimple/SSA*.

[11] `http://www.netlib.org/lapack`

[12] While a good numerical scientist would probably never write code like $M + M$ (where M is some big matrix) himself, some preprocessing or macro-expansion could bring such occurrences, and handling them inside the compiler can be worthwhile.

```
/* GIMPLE_RETURN <RETVAL> represents return statements.

   RETVAL is the value to return or NULL.  If a value is returned it
   must be accepted by is_gimple_operand.  */
DEFGSCODE(GIMPLE_RETURN, "gimple_return", GSS_WITH_MEM_OPS)

/* GIMPLE_PHI <RESULT, ARG1, ..., ARGN> represents the PHI node

   RESULT = PHI <ARG1, ..., ARGN>

   RESULT is the SSA name created by this PHI node.

   ARG1 ... ARGN are the arguments to the PHI node.  N must be
   exactly the same as the number of incoming edges to the basic block
   holding the PHI node.  Every argument is either an SSA name or a
   tree node of class tcc_constant.  */
DEFGSCODE(GIMPLE_PHI, "gimple_phi", GSS_PHI)
```

Figure 5: two cases of *Gimple* from `gcc/gimple.def`

```
/* All pointer-to-x types have code POINTER_TYPE.
   The TREE_TYPE points to the node for the type pointed to.  */
DEFTREECODE (POINTER_TYPE, "pointer_type", tcc_type, 0)

/* Contents are in TREE_REAL_CST field.  */
DEFTREECODE (REAL_CST, "real_cst", tcc_constant, 0)

/* Pointer addition.  The first operand is always a pointer and the
   second operand is an integer of type sizetype.  */
DEFTREECODE (POINTER_PLUS_EXPR, "pointer_plus_expr", tcc_binary, 2)

/* & in C.  Value is the address at which the operand's value resides.
   Operand may have any mode.  Result mode is Pmode.  */
DEFTREECODE (ADDR_EXPR, "addr_expr", tcc_expression, 1)

/* RETURN.  Evaluates operand 0, then returns from the current function.
   Presumably that operand is an assignment that stores into the
   RESULT_DECL that hold the value to be returned.
   The operand may be null.
   The type should be void and the value should be ignored.  */
DEFTREECODE (RETURN_EXPR, "return_expr", tcc_statement, 1)

/* Variable references for SSA analysis.  New SSA names are created every
   time a variable is assigned a new value.  The SSA builder uses SSA_NAME
   nodes to implement SSA versioning.  */
DEFTREECODE (SSA_NAME, "ssa_name", tcc_exceptional, 0)
```

Figure 6: some cases of *Tree* from `gcc/tree.def`

3 Some hints and advice for taming GCC with MELT

Extending GCC is challenging, mostly because of the complexity of compiler technology (so extending other industrial strength free software compilers like OPEN64 or LLVM/CLANG would also be difficult).

Customizing GCC (with MELT) raises several issues:

- understanding the intricated GCC internal representations, including *Gimple* instructions (i.e. the `gimple` data) and *Tree*-s (`tree` pointers), but also `gimple_seq` (a sequence of gimple instructions), `basic_block` (an elementary block of instructions, containing a `gimple_seq`, entered only at its start, and jumping to other basic blocks or returning to the caller function), `edge` (an arrow between `basic_block`-s, with GCC knowing each side of it), or even `function`-s or `cgraph`-s. Furthermore, all these data types are not available at all times in GCC (some passes are run when some data is not computed yet).
- understanding the numerous GCC internal passes (there are ≈ 250 of them!), and what kind of new passes should be written, and where to insert them. There are no obvious coding rules relating a pass appearing in the GCC source code, to its name.
- understanding the MELT domain specific programming language and its interface to GCC (if coding a GCC plugin in *C*, understanding the coding conventions and GCC plugin interface to *C* is also challenging).
- choosing what pass[es] should be added into GCC; the pass manager (in function `init_optimization_passes` from file `gcc/passes.c` of the GCC source code) is organizing the passes into several kinds (see file `gcc/tree-pass.h`)
  - `GIMPLE_PASS` for "simple Gimple passes" working on one function at a time;
  - `RTL_PASS` for back-end "Register Transfer Language" passes (which are target-processor specific, so probably not relevant here);
  - `SIMPLE_IPA_PASS` for "simple Inter-Procedural Analysis" passes;
  - `IPA_PASS` for full "Inter-Procedural analysis" passes.

  Passes are described by `struct opt_pass` data structures, containing the kind and name of the passes, and some function pointers, notably the *gate* function pointer (often null) which decides if the pass has to be executed, and the *execute* function which does the real job of the pass.

  A pass of a given kind can only be inserted near other passes of the same kind.
- choosing which GCC internal representations should be used; this choice also depend of the considered pass; for instance, *Gimple/SSA* is not always available, or on the contrary may be required before some pass.

3.1 Choosing the right passes

The order and set of executed passes (within some given compilation by GCC) depend upon the actual code compiled by GCC and also of the optimizations requested at compilation time (e.g. is different with `-O1` and with `-O3`). Passes whose name starts with a letter can give *dump files* (a textual file dumping some internal representation handled by the pass, e.g. the *Gimple* code). The GCC program option `-fdump-tree-all` produces all the dump files; unfortunately the name (e.g. `matmul.c.004t.gimple`) of the dump file is not very meaningful (because the contained number e.g. `004` is *not* a chronological ordering of passes).

A possible way of understanding the concrete passes executed by GCC is to add a MELT hook with the `register_pass_execution_hook` function provided by MELT. The registered closure gets the pass name and number and is invoked before each pass.

Most MELT extensions involve insertion of additional passes (coded in MELT). Here are some few guidelines for choosing where to insert such passes:

- if your pass needs only *Gimple* instructions without control flow graph, consider inserting it after the `gimple` pass.
- if your pass needs *Gimple/SSA* instructions, insert your pass after the `ssa` pass at least.
- if your pass needs inlining to have been done, consider inserting it after the `phiopt` pass
- if your pass requires GRAPHITE to have detected parallelizable loops, insert it after `graphite` pass
- etc ...

Choosing the right place to insert your pass is still a difficult issue, notably when interprocedural optimizations are required (or are useful).

3.2 Using the available representations

GCC has a lot of internal data types and representations : the `gengtype` utility, which generates[13] marking routines for the GCC garbage collector, handles more than 1900 types with `GTY` annotations. So GCC has a lot more internal representations than just `tree`-s and `gimple`-s.

MELT is in principle able to access any such `GTY`-ed data type: part of the MELT runtime is generated, notably the layout of boxed values (e.g. boxed `gimple` or `edge`

[13]The GCC compiler has more than a dozen of specialized internal *C* code generators : `gengtype` generating garbage collection marking routines; `genattr` generating attribute information in the back-end from machine description; `genautomata` for pipeline hazards, etc. Most generators are not available to plugins.

values) and their low level *C* support code (e.g. for allocating such values, and for copying or scanning them in the MELT garbage collector). Each GCC stuff known to MELT is described in a *C-type* instance: so the `loop` GCC data is described in a `ctype_loop` descriptor (giving the `:loop` "keyword" to annote stuff of that type in MELT code), etc. So adding a new *C-type* is a matter of providing its descriptor and then regenerating all MELT generated code (inside the MELT branch).

The GCC compiler does not provide any direct mean to add specific data *inside* existing representations (like `gimple`, etc.). To associate their own data to such existing representations, GCC plugins and MELT extension have to manage this association outside of the internal representations. A convenient way is to keep in some hash-table, keyed by GCC representations like `gimple`, `tree`, etc the association between internal GCC representations and MELT (or GCC plugins') extensions. Hence, every *C-type* provides not only boxed values (so a boxed *Tree* is a MELT value containing a `tree` pointer), but also homogeneous hash-tables: e.g. a map of *Tree*-s is an hash-table, itself a MELT value, having `tree`-s as keys, and arbitrary non-null MELT values associated to each of them.

With the *Gimple/SSA* representation, GCC manage also the "use-def" information: when $trssa$ is some raw `tree` of an *SSA* name from some *Gimple/SSA* instruction, the `walk_use_def_chain_depth_first` and `walk_use_def_chain_breadth_first` MELT primitives apply a given MELT closure to a given value and to every `tree` and `gimple` defining (i.e. setting) or using the variable under $trssa$.

4 Conclusion and future work

This paper should suggest developers of important scientific high performance computation software to consider extending and customizing the GCC compiler suite for their specific needs. This could in particular be useful to take advantage of GPGPUs and other specialized hardware.

Diving into GCC is an intimidating experience (because of the complexity and size of the `gcc` legacy softwware), but this paper gave some practical hints to make it easier.

Extending or customizing GCC has the decisive advantage of leveraging on the powerful internal representations and transformations already provided by GCC. These assets are the main reasons to build your extensions above GCC. In addition, extending GCC is adequate when implementing custom optimizations or code transformations.

Future work on MELT (if compatible with available funding) may include interfacing additional GCC features, more OPENCL related extensions, a Web interface to MELT, analysis to help drive energy-consumption related facilities.

High performance computing is a major potential application of compiler customizations. MELT extensions should be "easily" prototyped to implment such specific enhancements.

Acknowledgments

Work on MELT has been possible thru French public funding from DGCIS of GlobalGCC and OpenGPU projects.

References

[1] A.Kravets, A. Monakov, A. Belevantsev, "Graphite-OpenCL : Generate OpenCL code from Parallel Loops", in *GCC Summit*, 2010.

[2] C. Augonnet, S. Thibault, R. Namyst, P.A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures", *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23: 187–198, Feb. 2011, `http://hal.inria.fr/inria-00550877`.

[3] G. Fursin, O. Temam, "Collective optimization: A practical collaborative approach", *ACM Transactions on Architecture and Code Optimization*, 7: 20:1–20:29, December 2010, ISSN 1544-3566, `http://doi.acm.org/10.1145/1880043.1880047`.

[4] GCC, "GCC internals doc.", march 2012, `http://gcc.gnu.org/onlinedocs/gccint/`.

[5] K. Karimi, N.G. Dickson, F. Hamze, "A Performance Comparison of CUDA and OpenCL", *Read*, cs.PF(1): 12, 2010, `http://arxiv.org/abs/1005.2581`.

[6] J. Pitrat, *Artificial Beings (the conscience of a conscious machine)*, Wiley / ISTE, march 2009, ISBN 9780470611791.

[7] B. Starynkevitch, "MELT code [GPLv3] within GCC", `http://gcc-melt.org/` and `svn://gcc.gnu.org/svn/gcc/branches/melt-branch`, 2006-2012.

[8] B. Starynkevitch, "Middle End Lisp Translator for GCC, achievements and issues", in *GROW09 workshop, within HIPEAC09, `http://www.doc.ic.ac.uk/~phjk/GROW09/`*. Paphos, Cyprus, january 2009.

[9] B. Starynkevitch, "MELT - a Translated Domain Specific Language Embedded in the GCC Compiler", in *DSL2011 IFIP conf.* Bordeaux (France), Sept. 2011, `http://adsabs.harvard.edu/abs/2011arXiv1109.0779S`.

[10] K. Trifunovic, A. Cohen, "Enabling more optimizations in GRAPHITE: ignoring memory-based dependences", in *Proceedings of the 8th GCC Developper's Summit*. Ottawa, Canada, Oct. 2010, `http://hal.inria.fr/inria-00551509`.

[11] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin, R. Upadrasta, "GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation", in *GCC Research Opportunities Workshop (GROW'10)*. Pisa, Italy, Jan. 2010, http://hal.inria.fr/inria-00551516.