Exposing Complex Numbers to Target Back-ends

Sylvain Noiry
Kalray SA, Montbonnot Saint-Martin, France
snoiry@kalrayinc.com

Paul Iannetta
Kalray SA, Montbonnot Saint-Martin, France
piannetta@kalrayinc.com

Abstract

Complex numbers are used to describe many physical phenomenons and are of prime importance in data signal processing. Nevertheless, despite being part of the C and C++ standards since C99, they are still not completely first class citizens in mainstream compilers. This can be explained by the fact that most general purpose instruction sets did not have hardware support for complex operations for a long time. Here, we (1) review the current state of complex numbers in GCC and LLVM; (2) explain why supporting complex numbers in a generic and extensible way that can be finely tuned for each architecture is important; (3) present a framework which extends complex numbers support in GCC, and allows each target to specify how complex numbers should be lowered, how they can be vectorized, and how operations on complex number should be mapped to assembly instructions. We conclude by presenting a before/after comparison on a selected set of programs and suggest some ideas to improve the framework that we would like to consider next.

1 Introduction

One way to improve performance is to identify common patterns and build hardware with dedicated instructions. Digital Signal Processing and other areas of Physics make heavy use of complex numbers (most notably when computing Fourier transforms). To our knowledge, the APEmile was one of the first chip to come with native support in 1996. Currently, we can also note that Hexagon provided native support for operations on complex numbers. And with the evolution of vector extensions, other manufacturers have recently introduced instructions which can operate directly on complex numbers as well. However, GCC's current support of complex number does not expose complex numbers to target back-ends which precludes the selection of native instruction working on complex numbers.

1.1 GCC and Complex Numbers

The support of complex operation in GCC has started even before the adoption of C99, during the 1990s. One important aspect from this period which remains today is the split of a complex element in two pseudo registers grouped in an element called CONCAT that can be non-contiguous allocated during the ***expand*** pass [1]. This pass transform the GIMPLE target-independent intermediate representation into the RTL target-dependent one. This approach relaxes the register allocation on targets with small register files, like IA-32. However, there have been multiple attempts to replace the CONCAT by contiguous hard registers [2] or more advanced structures [3]. One of the reason is the arrival of processors with native complex instructions, such as the `APEmille` [4] in 1996. Because 32-bit float complex are the most common, performance issues associated with the extraction of the inner parts have also been identified on 64-bit processors [5, 6].

At the beginning, the lowering of complex operations was performed during the *expand* pass [5]. At that point, exposing patterns handling complex number to the RTL would have been possible. However, since at that time, no targets had native complex instructions, the lowering of complex numbers was moved into its own GIMPLE pass (`cplx_lower`) [7], when the GIMPLE representation was converted to the Single Static Assignment (SSA) around 2004.

Although there was some recent changes concerning the optimizations to the support of moves of complex types [8], the overall mechanisms are still the same. Now that native complex instructions are becoming more and more present in today's instructions sets [9, 10] this is starting to become an issue. ARM and Intel are using work around relying on pattern matching in the SLP vectorization pass to reconstruct operations on (vectors of) complex numbers from the lowered representations [11, 12]. However, due to the fact that it relies on pattern matching it might miss some opportunities and the current state of the implementation is not completely target-agnostic.

1.2 Related Work

Even though the multi layered IR (MLIR) has a means to deal with complex number through the complex dialect [15], LLVM does not expose complex numbers to the back-end writers either despite a sporadic but continuous demand since at least 2010. The last proposal [14, 13] reviews some past discussions and summarize the main use-cases (digital signal processing and geometry in the complex plane), as well as two directions to implement the support for complex numbers. The first one would rely on a new set of intrinsic functions, doing so would be mostly transparent and the new intrinsic functions could be ignored by parts which want to ignore complex numbers, however, it would make it harder to work on complex numbers and harness their algebraic properties. Another direction, which is also the one we will follow in the rest of the paper, is to add complex scalar and complex vector types into the compiler and expose them to the target back-ends. We think that commercial compilers such as ICC (before using LLVM as its core) and the IBM XLC compiler might have some support to ease the generation of efficient code dealing with complex numbers, but we were not able to find any references. If you have some, please feel free to contact us, and we will update this section accordingly.

1.3 Overview

Throughout this paper, most examples will use the Kalray Coolidge (Version 2) instruction set since it has native instructions for complex numbers as well as SIMD operations on them. However, the changes to GCC that we present here are target-agnostic and gated behind target hooks.

First, we will present our goals, our design, and the rationale behind our design. In a second time, we will present the details of the implementation, highlight the minimal set of changes needed for a back-end to enable complex numbers support, and pinpoint other parts of the compiler already partially handled by GCC that could be integrated into our implementation. In a third time, we will show how those new features enable a gain in speed on a Fast Fourier Transform by comparing the assembly code before and after on the Kalray Coolidge Version 2 MPPA. Lastly, we will review the things that could have been different, or more convenient as a back-end writter and present some future work directions.

2 Rationale

Since GCC already supports complex numbers, the extension is designed as super-set of the current implementation and tries not to duplicate existing code. In the current GCC, complex numbers are split into their real and imaginary parts. Once split, the compiler can almost forget about complex numbers altogether. We decided to reuse the complex scalar types, as well as add complex vector types. The major difference is that complex numbers are not split when a target back-ends exposes operations which can directly work on complex numbers. Reusing existing types allows reusing most of the current optimizations and vectorization comes almost for free.

```
(define_insn "addsci6"
  [(set (concat:CSI
          (match_operand:SI 0 "register_operand" "=r")
          (match_operand:SI 1 "register_operand" "=r"))
        (plus:CSI (concat:CSI (match_operand:SI 2 "register_operand" "r")
                              (match_operand:SI 3 "register_operand" "r"))
                  (concat:CSI (match_operand:SI 4 "register_operand" "r")
                              (match_operand:SI 5 "register_operand" "r"))))]
  ""
  ;; Add a pair of words
  "adddwp %0 = %2, %4")
```

Figure 1: Concat-based patterns

```
(define_predicate "concat_operand" (match_code "concat"))

(define_insn "addcsi3"
  [(set (match_operand:CSI 0 "concat_operand" "=r")
        (plus:CSI (match_operand:CSI 1 "concat_operand" "r")
                  (match_operand:CSI 2 "concat_operand" "r")))]
  ""
  ;; Add a pair of words
  "adddwp %0 = %1, %2")
```

Figure 2: Concat-operand-based patterns

Back-ends with hardware support for native SIMD instructions for complex numbers should have a mean to declare such vectors directly from C or C++, and the internal layout of such vectors should be opaque so that each target can choose what fits it best.

3 Implementation

3.1 Two Approaches

Exposing concat to the rtl. One of the most immediate approaches is to allow CONCAT to filter through RTL and allow patterns to match against it. Despite the fact that the impact on the existing code was minimal and did not introduce changes to the GIMPLE, it comes with several issues: code manipulating complex constants was not optimal (spurious extractions/insertions or spilling); the vectorizer would need changes, and writing patterns with CONCAT is not completely evident. One way is to match against CONCAT as seen in Figure 1, this is far from optimal since this changes the arity of the patterns and the proxies (`%0`, `%1`, `%2`) make assumptions on the layout of the register file. Another way, is to resort to a `concat_operand` predicate as in Figure 2, this has the advantage to not blow up the arity of the patterns, but we lose access to what is under the CONCAT. For all this reason, we decided to abandon this idea and focused on the following approach.

Exposing complex values to the rtl. This time we promote the already existing complex modes (`CQI`, `CHI`, `CSI`, `CDI`, `SC` and `DC`) to fully supported modes which may hold native values instead of only a CONCAT of two real values. The lowering pass (*`cplx_lower`*) split complex numbers into components on demand, and the rest of the compiler passes work mostly as usual as if complex numbers were normal scalar values. Patterns such as addition, subtraction and other arithmetic operators are naturally extended to their complex counter-parts.

3.2 Conditional Lowering

Previously complex numbers were gradually lowered as see fit until the main lowering pass (*cplx_lower*) which splits all remaining complex numbers into their real and imaginary components. This makes perfect sense for instruction set without native support for complex numbers. However, it is show-stopper for instruction sets which do have them. It would be better to lower complex numbers to a pair of real numbers only when some operation is not supported. Hence, the need for conditional lowering. In this section we review what GCC has been doing until now and describes the changes we made to the front-end and to the middle-end.

Book-keeping omitted, complex numbers are currently processed in four steps: first, they are recognized (and simplified to some extent) at parse time; second, they are split by the *cplx_lower* pass; third, on some architecture the SLP (superlevel-word parallelism) pass recognizes and vectorizes operations on complex numbers to some extent; and finally, the *expand* pass transform GIMPLE to RTL. After those four steps, no operations on complex numbers exists anymore, except for function arguments and return value.

Parse time. Even though the major part of the lowering happens in the *cplx_lower* pass, some decisions, such as the detection of rotations (i.e., multiplication by the imaginary unit) are performed at parse time. This add noise that makes the job of the *cplx_lower* pass harder. Therefore, we decided to not split anything at parse time. Instead, we add annotations so that information discovered by the parser is forwarded to the *cplx_lower* pass.

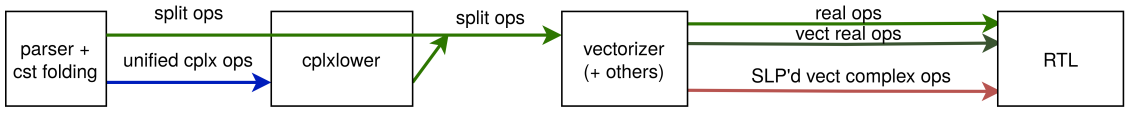
We should note that "complex numbers" are also used by some builtin functions with two return values (such as atomic compare exchange or overflow), but this does not prove to be a problem to our implementation.

The *cplx_lower* pass. This is where all operations on complex numbers are turned into operations on real numbers, or into a library call. Real and imaginary parts are extracted into `REAL_PART` and `IMAG_PART` and combined into a `COMPLEX_EXPR`. In some rare cases, these lowered operations will be caught when performing vectorization and generate SIMD instructions, but since most type information is lost even basic operations may be missed.

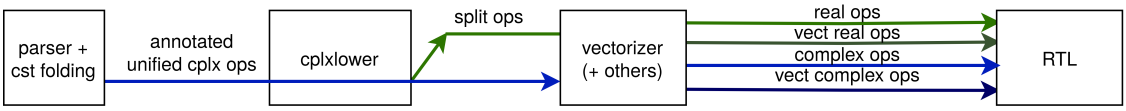
```
/* Before the (*@\cplxlower@*)pass */
_3 = a_1(D) + b_2(D);
/* After the (*@\cplxlower@*)pass */
a$real_5 = REALPART_EXPR <a_1(D)>;
a$imag_6 = IMAGPART_EXPR <a_1(D)>;
b$real_7 = REALPART_EXPR <b_2(D)>;
b$imag_8 = IMAGPART_EXPR <b_2(D)>;
_9 = a$real_5 + b$real_7;
_10 = a$imag_6 + b$imag_8;
_3 = COMPLEX_EXPR <_9, _10>;
```

On the other hand, our implementation keeps complex numbers as much as possible and resort to real and imaginary parts only when there is no target support and in some special cases. This is detailed in section 3.4 and section 3.3.

The *expand* pass. Until now, this pass translated complex numbers by creating a CONCAT structure made of two pseudo-registers holding the real and imaginary parts. The RTL that is produced directly uses the pseudo-registers inside the CONCAT except for function arguments and return values where the "complex register" associated with the whole CONCAT structure is used. Practically, this means that target back-ends never a get a chance to see registers with a complex type. The pseudo-registers referenced by a CONCAT structure are later mapped either to a single register (e.g., a complex float fits in a 64-bit register), or to multiple (non necessarily contiguous) registers. The real and complex parts are accessed in RTL via independant registers or subregs even for trivial operations, which oftentimes leads to spurious insertions and extractions. CONCATs were explicitly introduced to allow complex



(a) Old complex operations flow



(b) New complex operations flow

Figure 3: Complex operations processing in GIMPLE

types to be mapped to non-contiguous registers to ease register allocation on targets with a small register file [1], however, it incurs a performance loss on modern 64-bit targets where the insertion and extraction are much more frequent.

Our implementation behaves the same when it comes to complex numbers that have been split, however, it exposes three functions from the generic code as hooks (TARGET_GEN_RTX_COMPLEX, TARGET_READ_COMPLEX_PART and TARGET_WRITE_COMPLEX_PART), so as to give control over the creation of complex registers, and how complex types and complex vector types are stored and fetched from memory.

Figure 3 summarizes the differences between the old and the new behavior. In particular, the pass *cplx_lower* does not always lower complex numbers, and now the vectorizer also takes vectors of complex numbers as input.

3.3 Patterns for Complex Modes

Up until now, operations on complex numbers could not be described in machine descriptions. Hence, patterns were only generated for scalar modes (i.e., integers and floating-point numbers). Rather than expanding the list of optabs with new pattern names such as `cmulsi` for the multiplication of complex (32-bit) integers, we decided to extend the available modes for all arithmetic patterns. This way, the pattern for the multiplication of complex integers becomes `mulcsi`, and avoid duplicating standard optabs. Likewise, we extended legal vector modes to complex modes as well, which allows to describe SIMD operations on complex numbers, and enable the autovectorizer (cf Section 3.6).

Depending on whether a target fills those complex patterns or not the lowering pass (*cplx_lower*) will either keep the complex as-is or split it into its components. Currently, the supported `optab`s are: `mov`, `add`, `conj sub`, `mul` and rotations (`crot90` and `crot270`). This complex division `div` could also be added but it is not currently supported. This makes it easy to support the patterns added by ARM (`cmul`, `cmul_conj`, `cadd90` and `cadd270`). In fact, by providing a finer-grained implementation all those patterns can be recognized by our implementation and the combination of two simplex operations such as `cmul_conj` can be handled by the *combine* pass.

We keep the old behavior and split the complex numbers into a pair of real numbers in two cases: when a target does not have native support, or when either the real or imaginary part is a constant. The case where either the real or imaginary is a constant leads to decisions which are not clear-cut. In particular, we should be careful not to hamper constant folding and constant propagation even when the complex numbers are not split.

Fast patterns. Among all the complex types operations, a machine can probably not handle all of them natively. Sometimes, it could be more efficient to emulate an operation manually than letting GCC splitting it. However, writing an emulated pattern using `define_expand` or `define_insn_split` may not be easy when IEEE compliance needs to be checked a floating point type. `fast patterns` have been designed as a way to only write the fast path of emulated IEEE compliant operation. All checks are kept in the generic code, like in `expand_complex_multiplication` for the complex multiplication.

A \ B	uninit	real	imaginary	full complex
real	**lower**	**lower**	**native**	**native**
imaginary	**lower**	**native**	**lower**	**native**
full complex	**native**	**native**	**native**	**native**

Figure 4: Conditional Lowering

A back-end programmer can use it by creating a pattern prefixed by `fast_` before the optab name. `Fast patterns` can be interesting when dealing with emulated operation on any composite type.

The presence of both "unified" complex numbers and "split" complex numbers leads to some complexity that we will address in a later sections.

3.4 "Unified" & "Split" Complexes

Because all complex operations need not be supported by a target, some of them will be lowered. Moreover, there are some cases where it makes sense to lower them anyway even though native operations may exists. Thus, our implementation has to support a mix of both representations, and it should be simple to switch from one to the other.

Figure 4 details in which cases a complex operation on **A** and **B** will lower its operand or keep them as unified complex numbers. The cases where **B** is not initialized corresponds to unary operations on **A**. The gist of it is that when the operands are in fact both reals or imaginary numbers the operation is split otherwise we keep it as-is as long as the target supports it.

It should be easy to switch from both representation at any time, hence, complex constants and SSA variables can be accessed through three components: their real part, their imaginary part, or both of them. Of course, the inner real and imaginary part of the whole complex refers to the same components as any of the two separately (see Fig. 5. Thus, any change to the whole complex will update the components and any change to a component will update the whole complex as well. This means that the three components are always up to date, and can be used interchangeably. One drawback is that we generate instructions to update each component separately and it is clearly observable when optimizations are not enabled. As this is dead code, it is removed by optimizations, and we did not make any efforts to reduce unneeded instructions at `-O0`.

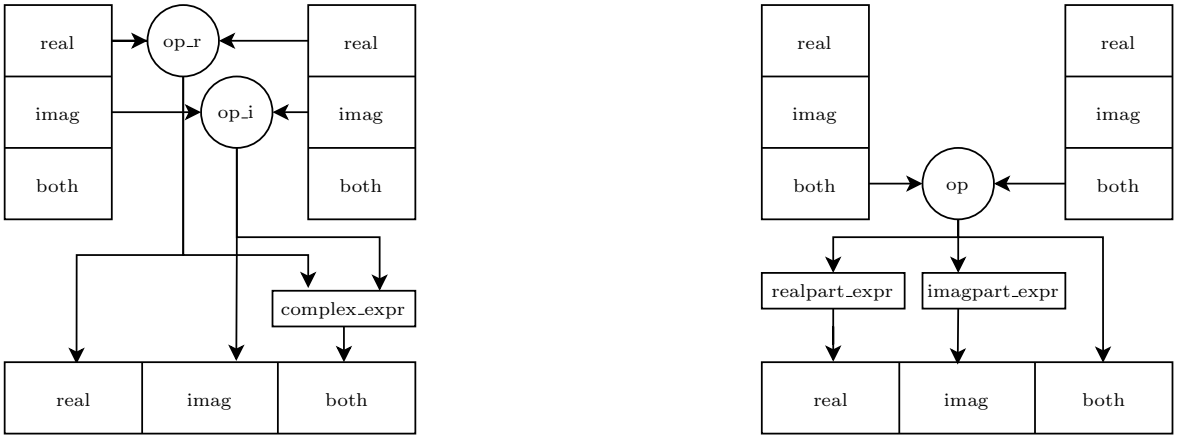
3.5 Fusion of Complex Operations

The design choices outlined above allows passes that merge operations into bigger ones, such as FMA detection or the *`combine`* pass, works well with complex modes and we are able to synthetize complex FMA or operations, such as rotate & add, or add & conjugate.

3.6 Vectorization

Currently, GCC's vectorization passes only accept vectors of scalar elements. ARM's attempt to bypass this restriction by introducing pattern matching in the SLP vectorization pass to infer complex operations from a series of operations on real number that looks like an operation on complex numbers.

This is sub-summed by our implementation in all cases except when the pattern matching synthesized a complex operation "by chance" even though the original source code was not working with complex numbers.



(a) Lowered operation

(b) Native operations

Figure 5: Mixing native and lowered complex operations

(a) Real and imaginary parts in separate vectors (SCV2)

(b) Real and Imaginary parts interleaved (V2SC)

Figure 6: A complex vector of length 2

Even though complex numbers cannot always be considered as scalars, "unified" complexes behave exactly like scalar elements from the point of view of the vectorizer. Henceforth, allowing the vectorizer to work on "unified" complexes is all that was needed to enable vectorization on complex modes. Recent discussions on the mailing list [16] highlighted that the representation of vectors of complex numbers is architecture dependent: exemple of such representations (SCVx and VxSC) can be seen in Figure 6. Both make sense depending on the target architecture and the different kind of loads and stores it exposes (especially the presence of interleaved loads and stores). Thus, the vectorizer does not make any assumptions on the layout of vectors of complex numbers, and defer everything to machine description.

We should note that this duplicate in part the features [11] behind the patterns dealing with complex numbers deguised as vectors of float and caught by pattern names starting with "c" such as `cmul`, `cmul_conj`, `cadd90`.

4 Harmonization

Currently the implementation still has some rough corners due to the fact that we wanted to not break anything, or introduce huge changes in the front and middle ends of GCC. However, once the new implementation will be merged we may want to make more deep changes.

- Some decisions related to complex lowering are handled elsewhere the complex lowering pass (*`cplx_lower`*). As this makes conditional lowering harder we suggest that passes that currently split complex numbers into their components instead add an annotations that will then be processed by the *`cplx_lower`* pass. This kind of approach has been successfully tested on complex rotations using internal functions.
- The pattern-matching code in the SLP pass, as well as the associated special handling of some cases needed for the pattern-matcher to work properly could be removed.

```
_Complex float
mul(_Complex float a, _Complex float b)
{
  return a * b;
}
```

(a) Complex multiplication

```
mul:
    copyw $r3 = $r0
    # extract imag part
    extfz $r5 = $r0,32+32-1,32
    ;;      # (end cycle 0)
    # float multiply
    fmulw $r4 = $r3, $r1
    ;;      # (end cycle 1)
    # float multiply
    fmulw $r2 = $r5, $r1
    # extract real part
    extfz $r1 = $r1,32+32-1,32
    ;;      # (end cycle 2)
    # float multiply substract
    ffmsw $r4 = $r5, $r1
    ;;      # (end cycle 5)
    # float multiply add
    ffmaw $r2 = $r3, $r1
    ;;      # (end cycle 6)
    # insert back real part
    insf $r0 = $r4,32+0-1, 0
    ;;      # (end cycle 9)
    # insert back imag part
    insf $r0 = $r2,32+32-1,32;
    ret
    ;;      # (end cycle 10)
```

(b) Before

```
mul:
    # Complex float multiply
    fmulwc $r0 = $r0, $r1
    ret
    ;;      # (end cycle 0)
```

(c) After

Figure 7: A complex multiplication

- Currently, since complex modes are not under the umbrella type `scalar_mode`, we need to provide a duplicate of the `simd_preferred_simd_mode`, and this hampers accepting vector of complex numbers since there is an explicit check whether the type is `scalar_mode` or not. This could be improved by relaxing the constraints on `scalar_mode`, or by introducing a dedicated list of authorized mode as inner vector types.

5 Results

5.1 Small examples

The benefits of a full support of complex operation in GCC can be seen even for very small code example. The example described in figure 7 is a simple precision complex multiplication. the code generated previously (figure 7b) for the KVX machine was inefficient. With this implementation (figure 7c), a single native instruction has replaced everything. Similar results are obtains for additions, subtractions, and negations.

```c
void
fmacplx (
  float complex a[restrict N],
  float complex b[restrict N],
  float complex c[restrict N],
  float complex d[restrict N])
{
  for (int i = 0; i < N; i++)
    d[i] = c[i] + a[i] * b[i];
}
```

(a) Complex vector FMA

```
fmacplx:
    make $r4 = 0
    make $r5 = 32
    ;;      # (end cycle 0)
    # begin hardware loop
    loopdo $r5, .L56
    ;;      # (end cycle 1)
.L53:
    # load 128-bit
    lq.xs $r10r11 = $r4[$r1]
    ;;      # (end cycle 0)
    # load 128-bit
    lq.xs $r8r9 = $r4[$r0]
    ;;      # (end cycle 1)
    # load 128-bit
    lq.xs $r6r7 = $r4[$r2]
    ;;      # (end cycle 2)
    # float complex FMA two lanes
    ffmawcp $r6r7 = $r10r11, $r8r9
    ;;      # (end cycle 5)
    # store 128-bit
    sq.xs $r4[$r3] = $r6r7
    addd $r4 = $r4, 1
    ;;      # (end cycle 8)
    # loopdo end
.L56:
    ret
```

(b) After

Figure 8: A complex vector FMA

In addition, fused or vectored operations have also been optimized. Considering a multiply accumulate operation of vectors of complex elements of the figure 8, the generated code is shown in 8b. The complex multiplications and additions have been fused. The loop has also been vectorized using SIMD complex FMAs (`ffmawcp`). Memory operations have also been grouped by packets of 128-bit.

5.2 A radix 2 Fast Fourier Transform

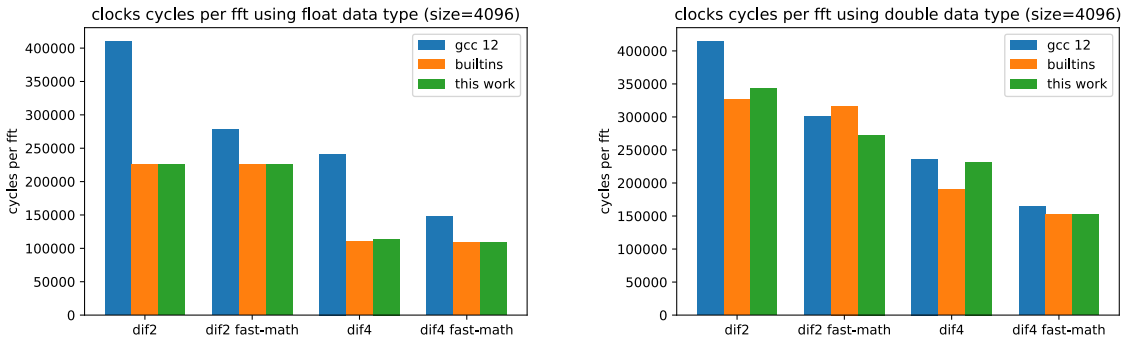
Fast Fourier Transforms (FFT) are one of the most popular type of algorithms which uses complex numbers in their inner logic. Until now, most implementations (cf papier benoit) redefines and uses complex types and operations rather the standard C complex ones because of the poor performance of the compiled code. Thus, a radix-2 in-place implementation has been used for benchmarking against a standard GCC and another version that redefined complex operations using builtins. The code used for this benchmark is in the listing of Figure 9.

Figure 10 shows the execution time obtained for the three different scenarios.

```
#define CMUL(a, b) __builtin_kvx_fmulwc(a, b, "")
#define CFFMA(a, b, c) __builtin_kvx_ffmawc(a, b, c, "")

void burrus_dif2_fft_float_(long n, cxf_t x[], float dir) {
  long l2n = __builtin_ctzl(n);
  for (long k = 0; k < l2n; k++) {
    long n1 = n >> k;
    long n2 = n1 >> 1;
    cxf_t s1 = stage_dif2_fft_float_steps[l2n - k];
    s1[1] = copysignf(s1[1], -dir);
    cxf_t e1 = { 1.0, 0.0 };
    for (long r = 0; r < n/2; r++) {
      long j = r >> k;
      long i = j + ((r<<(l2n-k)) & n-1);
      long l = i + n2;
      cxf_t t = x[i] - x[l];
      x[i] = x[i] + x[l];
      x[l] = CMUL(t, e1);
      cxf_t e1n = CFFMA(e1, s1, e1);
      if (j != (r+1)>>k) {
        e1 = e1n; } } } }
```

Figure 9: DIT Radix-2 FFT



(a) Complex float data type (FP32)

(b) Complex double data type (FP64)

Figure 10: Benchmarks results on FFTs

6 Implementation Summary

The approach described in this article has been implemented in the fork of GCC 12 maintained at Kalray. It consists of a series of 13 patches, which are publicly available in a dedicated branch [1]. Some of them add the core features of the new way of processing complex number, whereas some others introduce optimizations and side features. Back-end specific patches finally exploit these features for the KVX target.

6.1 Current State

At the core of the implementation is the conditional splitting of complex operations depending on the existing patterns in the target back-end. This allows natively supported operations to go to the *expand* pass. Then, the three hooks which allow each back-end to have its own representation of complex elements in RTL are the second major feature, as well as multiple update in the *expand* pass to allow more diversity with complex handling.

Some complex specific operations have also been optimized. One of these is the introduction of a conjugate operation in the RTL and in the machine description, which expands a GIMPLE CONJ_EXPR. The way complex rotations are handled has also been improved to work without SLP pattern matching. Complex FMA can be generated like with any real type, whereas other fused instructions utilizes the *combine* pass.

The implementation also add vectors of complex and provide an experimental support for explicit and automatic vectorization. However vectors of composite types are not accepted in upstream for now, but discussions are not closed. Fast patterns, as described in section 3.3, have also been originally introduced to emulate the complex double multiplication without manual checking of IEEE compliance.

6.2 Future Work

Future works mainly involve the harmonization of the current implementation with the choices made for the future of GCC. Assuming that the approach described in this article will not drastically change, some points could be improved.

The experimental support for complex vectors has to be stabilized. It could also be interesting to improve the auto-vectorization capabilities of mixed "Unified" and "Split" operations. Because different vector types are used, this makes the vectorization of these more difficult. Explicit vectors of complex type have also been experimentally added (using `__attribute__((vector_size()))`, but the implementation is currently not stable enough, mainly because the *cplx_lower* pass cannot handle vectored operations for now. One solution could be to lower explicit vectors by calling the `vec_lower` pass before *cplx_lower*.

More outside complex numbers, the `fast patterns` which have been introduced in this implementation may also benefit to other operations, especially when emulating composite operations.

7 Extend Back-ends features

This section is quite different from the rest and focus mainly features that we think would be nice to have to allow back-end to retarget the compiler more effectively.

Currently, despite having some control, target back-ends cannot easily extend the langage dialect they are supporting, and cannot easily design custom passes which need to extend the features of the middle-end. For example, even though, it is possible to have a target-dependent `passes.def` file, it would be also nice to allow other `def` files to have a target-dependent counterpart.

[1] https://github.com/kalray/gcc/tree/complex/kvx

8 A minimal back-end

This section presents the main target hooks, and patterns introduced by our implementation and how to implement them to enable complex mode support in an existing back-end.

8.1 Target hooks

As explained in section 3.2, each machine can choose its own representation of complex elements in RTL by defining three target hooks:

- TARGET_GEN_RTX_COMPLEX: This hook returns an RTL element corresponding to a complex. Real and imaginary parts are passed by argument. If both are `NULL`, a newly created complex must be returned.
- TARGET_READ_COMPLEX_PART: This hook returns a specified component of a complex element. It can be the real part, the imaginary part, or both of them. The implementation can be inspired by the previous READ_COMPLEX_PART generic function.
- TARGET_WRITE_COMPLEX_PART: This hook writes a complex component into a complex element. The implementation can be inspired by the previous WRITE_COMPLEX_PART generic function. Note that a CONCAT type can still be passed in the field `val` even if the back-end uses an other type for complex elements. This is a hack meant to allow optimization of moves from expanded `COMPLEX_EXPR`.

8.2 Complex Patterns

The next steps consists of implementing complex patterns, like any other pattern. The used mode depends on the chosen complex representation in the above hooks. For common instructions with real modes, like a complex addition and a two lanes vector addition, some patterns can be defined as wrapper to already present ones. Don't forget to include `mov` patterns to fully exploit the implementation. For example, a basic set of patterns can include `mov`, `add`, `sub`, `neg`, `conj`, `mul`, `crot90`, and `crot270`. If some of these operations are not not natively supported by the target, emulated patterns can sometimes bring more performance than automatic lowering in *`cplx_lower`*. Indeed *fast patterns* can be useful if there is a need to check for IEEE compliance of an emulated floating point operation.

8.3 Enable Vectorization

To enable the vectorization of "unified" complex operations, the VECTORIZE_PREFERRED_SIMD_MODE_COMPLEX target hook has to be implemented. It is basically the complex counterpart of VECTORIZE_PREFERRED_SIMD_MODE. In addition, vector of complex modes have to be defined in the machine back-end, as well as vector patterns.

9 Aknowledgement

This research has received funding from the European High Performance Computing Joint Undertaking (JU) under Framework Partnership Agreement No 800928 (European Processor Initiative) and Specific Grant Agreement No 101036168 (EPI SGA2), and also from bpifrance i-demo program.

10 Conclusion

We presented a flexible implementation of complex numbers into GCC that reuse the already available machinery and enables target back-ends to fill-in patterns corresponding to complex numbers. This reuses the existing machinery and, as such can benefit from all others passes.

References

[1] GCC source code, *Richard Stallman*, 1993:
"For complex modes, don't make a single pseudo. Instead, make a CONCAT of two pseudos. This allows noncontiguous allocation of the real and imaginary parts, which makes much better code. Besides, allocating DCmode pseudos overstrains reload on some machines like the 386.",
Commit: 3b80f6ca69f7

[2] **consecutive hard regs in complexes**,
GCC mailing lists, *Benedetto Proietti*, 2000:
https://gcc.gnu.org/pipermail/gcc/2000-June/049041.html

[3] **generic PATCH: Replace CONCAT with RECORD_REGS**,
GCC mailing lists, *Greg McGary*, 1999:
https://gcc.gnu.org/pipermail/gcc-patches/1999-October/020122.html

[4] **APEmille: a parallel processor in the teraflop range**,
E. Panizzi, 1996:
https://arxiv.org/pdf/hep-lat/9609010.pdf

[5] **The complex problems**,
GCC mailing lists, *Jeffrey A Law*, 1999:
https://gcc.gnu.org/pipermail/gcc/1999-July/033369.html

[6] **complex float support and 64 bit hosts**,
GCC mailing lists, *Jim Wilson*, 1999:
https://gcc.gnu.org/pipermail/gcc-patches/1999-August/015609.html

[7] **[tree-ssa] lower complex operations**,
GCC mailing lists, *Richard Henderson*, 2004:
https://gcc.gnu.org/pipermail/gcc-patches/2004-January/124515.html

[8] **[PATCH] Complex move by parts**,
GCC mailing lists, *David Edelsohn*, 2005:
https://gcc.gnu.oimplmrg/pipermail/gcc-patches/2005-March/164182.html

[9] **Arm SVE documentation**:
https://developer.arm.com/documentation/ddi0584/latest/

[10] **Intel AVX512-FP16 documentation**:
https://networkbuilders.intel.com/solutionslibrary/intel-avx-512-fp16-instruction-set-for-intel-

[11] **[PATCH] middle-end: Support complex Addition**,
GCC mailing lists, *Tamar Christina*, 2020:
https://gcc.gnu.org/pipermail/gcc-patches/2020-November/559908.html

[12] **[PATCH]Arm: Add NEON and MVE complex mul, mla and mls patterns.**,
GCC mailing lists, *Tamar Christina*, 2021:
https://gcc.gnu.org/pipermail/gcc-patches/2021-January/564015.html

[13] **RFC: Complex in LLVM**,
LLVM forum, *David_A_Greene*, 2019:
https://discourse.llvm.org/t/rfc-complex-in-llvm/52400

[14] **Complex proposal v3 + roundtable agenda**,
LLVM forum, *David_A_Greene*, 2019:
https://discourse.llvm.org/t/complex-proposal-v3-roundtable-agenda/53439

[15] **C representation of complex types?**,
LLVM forum, *dpotop*, 2020:
https://discourse.llvm.org/t/c-representation-of-complex-types/2180

[16] **[PATCH] Add COMPLEX_VECTOR_INT modes**,
GCC mailing lists, *Andrew Stubbs*, 2023:
https://gcc.gnu.org/pipermail/gcc-patches/2023-May/619825.html