Leveraging OpenMP 4.5 Support in CLANG for Fortran

Hyojin Sung(✉), Tong Chen, Zehra Sura, and Tarique Islam

IBM Research, Yorktown Heights, USA
hsung@us.ibm.com

Abstract. Modern computer systems are increasingly parallel and heterogeneous, and the demand for high-level programming interfaces for such systems is rapidly growing. OpenMP 4.0 extended its CPU-based directives to support device offloading. Programmers can now simply insert directives to identify computations and data to be offloaded. Compilers/runtime with OpenMP support then manage code translation and data transfers. While there are various ongoing efforts to support OpenMP device offloading for Fortran as well as C/C++, the most widely used open-source compiler, LLVM, supports C/C++ only. In this paper, we describe our project, XLFLANG, that aims to build an OpenMP Fortran compiler by bridging an existing Fortran front-end and LLVM C/C++ front-end (CLANG). We translate output from IBM XL Fortran front-end into CLANG AST and feed it to CLANG where OpenMP directives are lowerized to LLVM IR. This approach allowed us to reuse CLANG code generation and LLVM optimizations while handling Fortran-specific features in our XLFLANG. However, language dependences of CLANG AST especially with OpenMP directive representations pose unique challenges both in correctness and performance aspects. We addressed these challenges to generate CLANG AST, taking care to expose possible optimization opportunities. We were able to map all major OpenMP offloading directives/clauses from Fortran to CLANG AST, and our evaluation shows the resulting AST does not add significant overheads or interfere with later optimizations.

1 Introduction

Modern computer systems are increasingly parallel and heterogeneous, and parallel programming to exploit available parallelism has become the norm for programming itself. However, parallel programming is historically known to be error-prone and difficult to maintain, and there has been continuous effort to make parallel programming more accessible and tractable. Providing high-level abstractions and structured control for parallelism is one of the successful approaches to improve the programmability and portability of parallel programs. Especially with heterogeneous systems that commonly have a general-purpose host offloading computations to special-purpose accelerators, high-level interfaces can efficiently hide low-level details of host-device communications. It also

B.R. de Supinski et al. (Eds.): IWOMP 2017, LNCS 10468, pp. 33–47, 2017.
DOI: 10.1007/978-3-319-65578-9_3

improves code portability so that the same program can run on systems with different accelerators.

As the demand for high-level programming interfaces for heterogeneous systems grows, various attempts have been made in academia and industry. OpenMP is one of the popular parallel programming models that extends its CPU-based interfaces to support device offloading. OpenMP provides high-level directives that programmers can insert at appropriate points to identify computations and data to be offloaded to devices. Then compilers/runtime with OpenMP support translate these directives to actual codes that transfer data and manage offloaded computations.

OpenMP currently provides programming interfaces for C/C++ and Fortran with offloading support. Fortran is a language especially strong with numeric computation and scientific computing. It provides rich array notations that enables various array and loop based optimizations. There is a large volume of Fortran programs and libraries accumulated for decades in high-performance computing areas. Such computationally intensive Fortran programs have strong potential to scale very well on massively parallel devices such as GPUs, allowing OpenMP compilers/runtime to seamlessly map OpenMP constructs to GPU kernels. However, to our knowledge, OpenMP compilers with full offloading support exist only for C/C++, but not for Fortran yet.

In this paper, we describe our project to provide OpenMP Fortran support with full offloading features and competitive performance. Our project, XLFLANG, aims to build an OpenMP Fortran compiler by bridging an existing Fortran front-end and LLVM C/C++ front-end (CLANG). Our translator takes output from IBM XL Fortran front-end and translates it into CLANG AST but using our own semantic analyzer. Once the AST is generated, it is fed into CLANG code-generation (CodeGen) and translated into LLVM IR format. Our key observation in XLFLANG design is finding a right translation level that maximizes the reuse of existing C/C++ OpenMP support while retaining and utilizing Fortran-specific information for efficient code generation. With this approach, we can incorporate Fortran-specific handling of OpenMP directives in XLFLANG semantic analysis phase (while common codes with C/C++ are reused from CLANG) and avoid repeating low-level code generation and low-level optimizations for LLVM IR.

There were various challenges in mapping Fortran to an AST designed for a different language for OpenMP support. While Fortran and C/C++ share many common features, they are meaningfully different in user-defined type representation and address-based type handling. This poses a major challenge for XLFLANG in handling Fortran-specific data structures such as dope vectors and common blocks correctly when they appear in OpenMP data clauses. In addition, call-by-reference function call semantics, extra alias information available in Fortran only, and other base language differences from C require XLFLANG to perform additional AST node generation or modification to map Fortran to valid CLANG AST. We aimed to address these challenges in XLFLANG by generating compatible CLANG AST, taking care to expose possible optimization opportunities. We found that this task is more challenging than expected because we do not introduce any Fortran-specific changes in CLANG.

Despite some limitations, we could map all major OpenMP directives and clauses from Fortran to CLANG AST. Evaluation shows that the resulting CLANG AST from XLFLANG does not add significant overheads or interfere with LLVM back-end optimizations, providing comparable performance to equivalent C versions.

The contributions of XLFLANG translator can be summarized as follows:

– XLFLANG provides full-feature OpenMP 4.5 support for Fortran by leveraging CLANG, the first open-source compiler with full OpenMP support with offloading directives.
– XLFLANG provides comparable performance to equivalent C benchmarks tested, showing that its Fortran-to-CLANG AST translation introduces manageable overheads and does not interfere with later optimizations.
– XLFLANG revealed new use-cases of OpenMP offloading directives and clauses in Fortran and contributed to expanding the specification.

The rest of the paper is organized as follows: Sect. 3 describes the major challenges we had with translating OpenMP features in XLFLANG: OpenMP data handling clauses. Section 4 discusses how the base language differences between Fortran and C/C++ affected our implementation: handling OpenMP atomic/reduction clauses with logical equivalence operators, linking global symbols, and utilizing alias information. In Sect. 5, we present our experimental results for XLFLANG on several kernels and two benchmarks. The paper wraps up with related work (Sect. 6) and conclusion (Sect. 7).

2 XLFLANG Overview

LLVM compiler provides a robust C/C++ front-end, CLANG, with full OpenMP 4.5 support, but it currently does not have comparable solutions for processing Fortran programs. Our approach to provide the same OpenMP support for Fortran with minimal effort is bridging an existing Fortran front-end (for Fortran parsing/lexing and basic semantic analysis) and CLANG/LLVM (for code generation for OpenMP and back-end compilation). For the front-end, we leverage an existing and acknowledged, robust XL Fortran front-end (FFE) [6]. The FFE is a component of the proprietary IBM XL product compiler. It implements the full Fortran 2003 standard, and also support earlier versions including Fortran 95, Fortran 90, and Fortran 77.

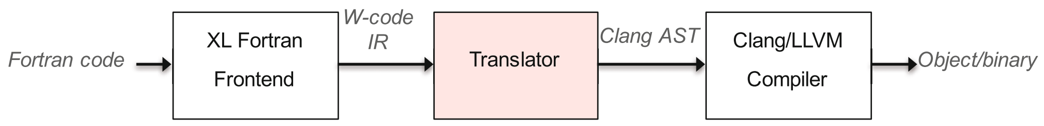

Fig. 1. The Overall Design of XLFLANG

Figure 1 illustrates the compilation flow of our translator. It transforms the output of FFE (W-code IR) into CLANG AST form, then the regular C/C++ CLANG/LLVM compiler is invoked with this AST as input.

2.1 XL Fortran Front-End

The FFE takes Fortran source code as input and produce W-code files as output. W-code is the intermediate format used within XL compiler. The front-end parses the source code and performs semantic analysis. Using the semantic analysis result, it augments user-provided OpenMP directives with necessary data handling clauses (details can be found in Sect. 3). It also includes a scalarize subcomponent that transforms Fortran array operations into corresponding loops. The W-code generated by the front-end contains very few Fortran language-specific features for example, array shape descriptors and some built-in functions, but it does include embedded information derived from language-specific semantics such as, for example, aliasing information.

2.2 XLFLANG

Our W-code to CLANG AST translator takes W-code as input and produces CLANG AST as output. It utilizes the CLANG Libtooling library [14] to parse command-line arguments, create a CLANG scope with appropriate initializations, populate it with AST code corresponding to W-code, and generate an AST binary file. The translator interfaces with an IBM-internal tool that decodes W-code binaries and provides utilities to iterate over the code and invoke user-defined actions for each W-code instruction. It first processes the entire W-code stream, gathering information and program elements used in the code, including types, literals, symbols, functions, labels, and static initializers. XLFLANG then once again traverses the code to generate the corresponding AST declarations, expressions, and statements. XLFLANG performs necessary transformations to the semantics of W-code output to generate CLANG-compatible AST, but does not modify control flows or loop structures.

2.3 CLANG/LLVM

The CLANG AST binary file produced by XLFLANG is then fed into the C/C++ CLANG/LLVM compiler to produce an object file or executable binary. XLFLANG relies on CLANG driver for compiling and linking the AST files with both Fortran and C/C++ libraries. Several XL Fortran libraries including Fortran I/O are linked by default. To link binaries and libraries with and without device support together, XLFLANG again relies on CLANG driver that allows linking a separate set of libraries for host and device objects.

3 OpenMP Data Handling Clauses with Common Block and Dynamic Variables

XLFLANG faced unique challenges in translating OpenMP features in the context of Fortran programs into CLANG AST. In many cases, translation is mechanical simply mapping an IR entry for an OpenMP directive/clause into a CLANG AST node of the same type. Unfortunately, we encountered numerous exceptions to this simple scenario due to differences between language features and IR specifications. In this and following sections, we focus on describing the major challenges addressed in XLFLANG for correct and efficient OpenMP support.

3.1 Background: Memory Objects in Fortran

The most common way to reference a memory object in Fortran is to use a variable name. Sometimes, scope, field, or subscript can be added. Unlike C/C++, pointer arithmetic with “address of” operator (&) or “dereference” operator (*) is not allowed. OpenMP data handling clauses specify how memory objects in the program should be allocated/freed, and how their value should be initialized for the program scope of constructs. These clauses include private, firstprivate, lastprivate, threadprivate, copyin, copyprivate, map, and reduction. CLANG performs semantic analysis to generate AST nodes for these clauses, but the semantics for Fortran variables in these clauses are not straightforward in some cases to simply reuse CLANG. With a simple example using a private clause for a scalar variable in Fig. 2, we illustrate how this clause requires different kinds of variables in Fortran to be treated.

```
integer a
```

(a) Local variable

```
subroutine mysub(a)
integer a
```

(b) Dummy parameter

```
integer s, a(100)
common /blk1/ a,s
```

(c) Common variable

```
integer, allocatable :: a
```

(d) Allocatable/pointer variable

Fig. 2. Different Variable Types in Fortran

Local Variable. In Fig. 2(a), the variable, `a`, is a local variable in Fortran, which is the counterpart of automatic variable in C/C++. The data handling clause can be directly translated into corresponding CLANG AST.

Dummy Parameter. In Fig. 2(b), the variable, `a`, is a dummy parameter of the procedure. Since parameters are passed by reference in Fortran (while by value in C), the access to parameter variable is a dereference from the pointer. In the W-code IR, there is a pointer, say `.a`, for the address passed through the parameter, and the references for s is actually `*.a` in IR. Consequently, the clause, `private(a)`, is actually `private(*.a)` in internal representation.

Common Variable. In Fig. 2(c), the variable, `s`, appears in a common block. In Fortran, variables in a common block are allocated contiguously and the common block with the same block names are linked across procedures. Common blocks in different procedures can have different layout of the variables. Therefore, variables in a common block are not independent variables, and are treated as array sections of the owner common block variable. In this example, the semantics of clause `private(s)` is `private(blk1[400, 4])`, where blk1 is the character array for the common block, 400 is the offset in the common block for variable s and 4 is the length. The array section is in C/C++ format, i.e., [start index : length] instead of [start index : end index].

OpenMP 4.0 standard does not allow array section in private clauses or copyin clauses for threadprivate. Such clauses in Fortran can not be directly translated into CLANG AST. Even though array sections are allowed in map clauses, we may still have issues when more than one variable in the same common block appears in the map clauses at the same construct. We do not want to map the whole common block for correctness and efficiency reason, but mapping each variable individually may result in multiple sections from one root variable, like `map(tofrom:a[0, 100], a[200, 50])`. This is currently not allowed by the OpenMP standard, requiring XLFLANG to find its own solution.

Dynamic Variable. There are three kinds of dynamic variable in Fortran, allocatable, pointer and assumable size array. Their size and shape are determined at runtime. To correctly access them, metadata about these variables are stored in a data structure, called "dope vector". Dope vector contains a pointer pointing to the data, a field for the status and fields for boundary of each dimension when the dynamic variable is allocated. Figure 2(d) shows an example of allocatable variable. When allocatable variables appear in a data handling clause, the action requested by the clause (data transfer, privatization, etc.) needs to be performed both on the dope vector itself and the data pointed by it. As a result, they cannot be mechanically mapped to CLANG AST.

In summary, the data handling clauses in Fortran require additional logic in XLFLANG to correctly initialize and transfer data, depending on the kind of the variable. In the following sections, we discuss our solution for the two major issues as identified above, array section and deep copy.

3.2 Transformation for Array Section

Example shown in Fig. 3(a) has a map clause for common variable *a* and `c`. XLFLANG translates variables `a` and `c` into array sections for the common block, and generates a map clause, `map(tofrom: blk[0, 100], blk[300, 400])`. There are two sections from a single root pointer blk. Such map clause is not allowed by OpenMP standard 4.0.

Our solution is to use separate temporary reference-type variables to represent each array section in the clause and in the enclosed code. Fortran allows such transformation, since all the data references are through the original variable, and can be easily identified for code transformation. In C/C++, the possible use of a pointer makes it difficult, if not impossible, to track the origin of each dereference. Moreover, overlapping may happen with pointer expressions. With the reference-type variable, different common variables are no longer referenced from the same root pointer. Each variables can be privatized or mapped individually. The transformation of the example is shown in Fig. 3(b).[1] Similarly, if a common variable is used in a private clause, we create temporary reference variables for each common variable and access them as an array (Fig. 4).

```
integer a(100), b(200), c(400)
common /blk/a, b, c
!$omp target map(tofrom: a, c)
do i = 1, 100
  c(i) = a(i)
enddo
```

(a) Original code

```
char (&c_ref)[1600] = *&blk + 1200UL;
char (&a_ref)[400] = *&blk;
#pragma omp target map(from: c_ref) map(to: a_ref) firstprivate(i)
{
   for (i = 1; i <= 100; i += 1) {
      ((int *)c_ref)[i] = ((int *)a_ref)[i];
   }
}
```

(b) Translated code

Fig. 3. Common Variable in `map` Clauses

```
integer a(100), b(200), c(400)
common /blk/a, b, c

!$omp parallel private(a, c)
  a(1) = 0
  c(1) = 2
```

(a) Original code

```
char (&c_ref)[1600] = *&blk + 1200UL;
char (&a_ref)[400] = *&blk;
#pragma omp parallel private(c_ref) private(a_ref)
{
  ((int *)a_ref)[0L] = 0;
  ((int *)c_ref)[0L] = 2;
}
```

(b) Translated code

Fig. 4. Common Variables in `private` Clauses

[1] The output is cropped from the result of CLANG ast-dump. Please be aware that the output of ast-dump from CLANG is in C format and may miss some necessary parenthesis. We use it in the paper because it is more readable than printing raw CLANG AST node information.

3.3 Transformation for Dynamic Variables

Because of the wide use of allocatable/pointer variable for dynamic data size in Fortran, handling dynamic variables correctly with deep copy is critical. As discussed above, "dope vectors" are used in Fortran to represent a pointer and other metadata about dynamic variables. When a dynamic variable appears in a map clause, we need to copy the dope vector and the data as well ("deep copy"), and fix the data pointer in the dope vector to point to the address of the newly mapped data. This may involves recursive traversal of multiple levels of pointers. The similar issue occurs for private clauses. When an allocatable variable appears in the private clause, first the data for the variable needs to be privatized. Secondly, the dope vector should be privatized so that each thread can allocate different size for the allocatable variable. More precisely, the dope vector should be firstprivate so that metadata can be customized per OpenMP thread. Therefore, XLFLANG needs to extends its deep copy mechanism for map clauses to the other data handling clauses too. Even though the problem is similar for map and private clause, they are handled differently with the interface provided in CLANG. Examples of map and private are shown in Fig. 5.

```
integer, allocatable :: mya(:), myb(:)


!$omp target map(to:mya) map(from:myb)
...
```

(a) Original code

```
char d_mya[56];
char d_myb[56];
char *_3, *_4, *_5, *_6;
#pragma omp target map(from: _5[0:_6]) map(to: d_myb)
          map(to: _3[0:_4])
{
   *&d_myb = _5;
   *&d_mya = _3;
}
```

(b) Translated code

Fig. 5. Allocatable Array in `map` Clause

In the example of Fig. 5(a), `d_mya` and `d_myb` are the dope vector for the allocatable array `mya` and `myb` in W-code IR, respectively. In the translated code, temporary pointers, _3 and _5, are initialized with the data pointer in the dope vector if the variable is allocated, otherwise null value is the default (the null value is not used in runtime). The size of the array section, variable _4 and _6, are also initialized from the dope vector. The references inside the target region still retrieve the information from dope vector to construct the access expression: the data pointer and the lower bound of the array.

Deep copy is a challenging issue in general for both Fortran and C/C++. OpenMP 4.0 supports only the most simple form of deep copy, array section with a pointer. More recent OpenMP 4.5 adds the support for pointer as a field, and as a result, temporary pointer and the assignments are no longer needed. XLFLANG implementation will be simplified once support for the latest standard is fully implemented. For private clauses, OpenMP standard the array section is not allowed. Therefore, we have to use reference-type variable for the data.

3.4 Combined Cases

In the example shown in Fig. 6, the variable `mya` is a parameter for allocatable array, which is passed in to the subroutine as a pointer to the dope vector. In our translator, we have to handle both the deep copy and array section. Consequently, one private clause in Fortran is actually translated into three private clauses in CLANG, as shown in the example.

```
subroutine test(mya)
integer, allocatable :: mya(:)


!$omp parallel private(mya)
...
```

(a) Original code

```

char (&d_mya_ref)[56] = *.d_mya;

char (&mya_ref)[*d_mya_ref + 48 * (*d_mya_ref + 40)] = *(*(d_mya_ref + 0));
#pragma omp parallel private(mya_ref) firstprivate(d_mya_ref)
         firstprivate(.d_mya)
{
  .d_mya = d_mya_ref;
  *d_mya_ref + 0 = mya_ref;
```

(b) Translated code

Fig. 6. Parameter Allocatable Array in `private` Clause

3.5 Limitations

We found that not all the cases of data handling clauses can be represented with CLANG AST. One example is copyin for threadprivate. When an individual common variable, instead of the common block name, appears in the copyin clause, there is no way to express this case with CLANG AST. This is a case of array section. However, neither array section nor referenced-type variables are allowed for copyin. The underlying reason is that the copyin needs the original symbol to find the corresponding threadprivate copy. Clang has to be extended to handle this case.

4 Challenges from Language Differences

In this section, we discuss how differences between Fortran and C/C++ at the language specification level influenced XLFLANG design and implementation.

4.1 Fortran-Only Operators in OpenMP Atomic and Reduction Directives

Fortran provides logical equivalence/non-equivalence operators (`.EQV.` and `.NEQV.`) that return true/false if and only if both operands have the same value respectively. C/C++ equality/non-equality operators (`==` and `!=`) offer the same logic. While these relational operators can replace `.EQV.` and `.NEQV.` in the plain non-OpenMP Fortran context, they cannot be used in OpenMP atomic or reduction clauses. Therefore, we emulate them with `XOR` operator (for `.NEQV.`) and the negation of `XOR` (for `.EQV.`) on arithmetic values. When `.EQV.` is used in OpenMP atomic and reduction clauses, additional code needs to be generated to perform the negation. In case of reduction, XLFLANG generates the reduction clause with `XOR` operator, and adds post-processing codes to perform the negation on the reduced result. Handling `.EQV.` in atomic clauses is more complicated than reduction clauses, since OpenMP atomic clauses have restrictions on the type of operations allowed. We exploit a mathematical property about `XOR` and `XNOR` to solve the issue; The result of `XNOR` on two operands is equivalent to the result of `XOR` on one operand and the negative of the other operand. In summary, we can implement an OpenMP atomic clause with `v = v XNOR exp` as `v = v XOR (!exp)`.

4.2 Linking Global Symbols

The common practice for declaring and defining global symbols is different in Fortran and C/C++. In Fortran, symbols for module variables do not appear in its object file, but .mod file is generated in addition to include metadata on module variables and functions. When the module variable is "used" by another module or function, the corresponding .mod file is referred and the variable appears in the using module's object file as a weak symbol. In summary, there does not exist a single object file with a strong symbol for module variables, but there can be multiple object files with weak or common symbols. When linked together, the final binary will resolve these weak symbols to point to the same storage. Globals can be linked in the same way in C/C++ by adding "weak" attribute ($\_\_attribute\_\_((weak))$) to each declaration/definition, but a much more common way is declaring it as "extern" in a header file and defining once in a source file.[2] In link time, one strong definition will generate a strong symbol in the binary (multiple definitions are not allowed).[3]

[2] Weak symbols are not mentioned by the C/C++ language standards.

[3] The symbol will be weak if uninitialized in C, or initialized to 0 in C++.

This different linkage style of globals caused an issue with compiling and linking module variables in `omp declare target` construct in XLFLANG. If a variable is `declare target` as shown in Fig. 7, the variable is automatically initialized on device as the program starts and can be accessed without explicit data transfer. It is useful when a variable is reused across multiple OpenMP target constructs.

```
module globals
...
!$omp declare target (mya, myb)
end module
```

globals.f

```
...
use globals
!$omp target
     mya = myb
!$omp end target
```

test.f

Fig. 7. Globals in `omp declare target`

When a module variable is `declare target` and accessed in other modules, the variable appears as a weak symbol (not an external reference) in each referring object file. OpenMP code-gen implementation in CLANG assumed each instance as a strong symbol and generated additional strong metadata symbols in each object file, causing multiple definition errors. Also, CLANG code-gen records all variables in `omp declare target` in a table and passes the table to OpenMP runtime to initiate data transfers. With multiple weak symbols from different object files, the table creates duplicate entries, and OpenMP runtime failed as it assumed a unique entry for each symbol. These cases in CLANG code-gen and OpenMP runtime are allowed by rare in C/C++, and the findings by XLFLANG drove fixes in CLANG that made OpenMP support and runtime more robust.

4.3 Intrinsic Aliasing Information in Fortran

Unlike C/C++, Fortran does not allow pointer arithmetic. Pointers in Fortran are just variables with the POINTER attribute, not a distinct data type. This leads to an important corollary that address aliasing through pointer arithmetic does not exist in Fortran. For XLFLANG to utilize the guarantee for non-aliasing addresses with minimal to no changes to other components, we used the `noalias` attribute for variables in CLANG AST. The keyword is originally for marking variables with `restrict` keyword in source codes. It later helps the alias analysis and other optimization passes in LLVM to build strong alias sets and determine the applicability of a given optimization. XLFLANG adds this keyword to function arguments for OpenMP outlined functions that capture code sections within OpenMP constructs. Alias information are often lost during the function outlining process if inter-procedural alias analysis cannot recover it, leading to many disabled common optimizations such as common subexpression elimination and LICM. Using the strong non-aliasing guarantee from Fortran, XLFLANG can safely add `noalias` attribute to arguments of OpenMP outlined functions, which cannot be trivially done for C/C++ programs.

5 Experimental Result

We evaluate the performance of XLFLANG by comparing the execution time of the "same" kernels written in Fortran and C compiled by XLFLANG and CLANG respectively. The evaluation was done on an OpenPower node using two Power 8 sockets (model PowerNV 8247-42L) and two NVIDIA Kepler GPUs K40m. The operating system run by the host processor is a bare-metal Linux distribution (Ubuntu version 14.04.1). The Fortran and C versions of the "same" kernel are intended to have the same operations to our best effort. There could be slight differences inevitably introduced by using different languages. We report the execution time of the computation kernels only to exclude possible differences in language libraries and setup.

We measure both the sequential performance of the kernels and parallel performance with OpenMP pragma to evaluate the performance on basic Fortran statements as well as OpenMP directives. We also gather data for the performance of kernels with gcc and gFortran as a control group. Since the gcc and gFortran share the same back-end optimization as CLANG and XLFLANG share LLVM back-end, the comparison can provide further insight of the performance of our system.[4]

All the reported execution times are normalized to the corresponding CLANG C performance for sequential or parallel version respectively. Wall time is measured for the sequential execution, while the data transfer time and kernel execution time acquired from nvprof are used for the parallel version.

The first kernel we used is simple vector_add. This kernel is used to evaluate XLFLANG for different variable types in Fortran, as discussed in Sect. 3. The second kernel we used is jacobi-2d from Polybench. Polybench provides both C and Fortran version, which is convenient for our experiment. We modified the kernels to add OpenMP directives for GPU offloading. It is straightforward for vector_add. For jacobi_2d, we add the `target data map` outside the nested loop and add `target teams distribute parallel do collapse(2)` to the two inner loop nests, as described in [10].

The execution time of sequential code is shown in Table 1. The performance for Fortran and C are almost the same when simple variables or parameter arrays are used. The difference between *c-simple* and *f-simple*, and *jacobi-2d-c* and *jacobi-2d-f* for CLANG/XLFLANG shows that the compilation overheads introduced by XLFLANG is minimal. There is a quite significant slowdown for common variables because the code is less efficiently scheduled due to the shared root pointer for the arrays. Allocatable array performs slightly better because it

[4] It is not the purpose of this paper to compare the sequential and the parallel performance. We did not aggressively optimize how the loops are parallelized for GPU. Nor is it the focus of this paper to compare the performance of CLANG and gnu compiler. For both compilers, we used -O3 for the sequential version only.

Table 1. Sequential execution time

kernels	CLANG/XLFLANG	gcc/gfortran
c-simple	1.00	1.5
f-simple	1.01	1.19
f-common	1.61	1.3
f-allocatable	0.88	1.07
f-parameter	1.01	1.06
jacobi-2d-c	1.00	1.38
jacobi-2d-f	1.05	1.38

Table 2. Parallel execution time

kernel time	HtoD time	DtoH time
1.00	1.00	1.00
1.03	1.00	1.00
1.03	1.0	1.00
2.69	1.02	1.00
1.26	1.03	0.99
1.00	1.00	1.00
1.01	1.00	0.93

triggered loop unrolling and doubled the unroll factor from 8 to 16. In the result for gcc/gFortran, it can be observed that similar trend for better performance on f-allocatable and worse performance on f-common. For the jacobi-2d kernel, XLFLANG introduced 5% slowdown while gcc and gFortran have the same performance on both versions of the code.

The result for offloading to GPU with OpenMP is reported in Table 2. The kernel execution time, the data transfer time for host to device (HtoD) and device to host (DtoH) are shown. Most of the Fortran code have the similar performance for kernel computation and data transfer, except for the vector_add_alloc. vector_add_alloc is more than 2.5 times slower than the other kernels. It is because the handling of the pointer in the dope vector disabled optimizations in CLANG code generation for OpenMP. When we move to the new OpenMP runtime interface, the dope vector will be handled directly with the support for pointer field in a struct. There will be no extra assignments in GPU code and the code can be optimized.

In summary, the experiments showed that XLFLANG is able to generate correct and comparably efficient binary for different Fortran variables, and with or without OpenMP directives.

6 Related Work

Many researchers and industry programmers proposed automatic conversion tools from Fortran to other modern languages such as C/C++, Matlab, and Python [1,4,8,9,12], and many of the projects are still active.

F2C [8] is one of the first Fortran to C source-to-source translator published in 90's. F2C prints out a C representation of the intermediate C parse tree by a Fortran 77 compiler. As a source-to-source converter, the tool uses C struct (and union of struct) and `#define` macros to represent Fortran common blocks and equivalence, which may significantly increase the resulting C code size. XLFLANG also takes intermediate IR (W-code) as input, but its translation takes place between intermediate language levels both for input and output. Translating to AST allows more efficient and succinct translation, circumventing various source-level limitations. Also, F2C works only for Fortran 77 files while XLFLANG is tested up to Fortran 2003 and much more robust with regard to Fortran specification changes as FFE lowerizes new features to common W-code IR.

FABLE [9] is a recent effort on automatic Fortran to C++ conversion. It is influenced by prior work [1,8], but applies various techniques to improve performance and readability of its C/C++ output including translating global variables and `SAVE` variables to C++ struct. It also supports a subset of Fortran 90 as well as Fortran 77. It requires iterative re-converting, compiling, and testing and manual code changes to improve performance and code quality of the final output. XLFLANG does not need iterative compilation and testing to get working AST, relying on LLVM for performance optimizations. Some of their optimization techniques including using C++ struct for global variables for modular binaries could be applied to future XLFLANG design.

Rose source-to-source compiler infrastructure [11] provides source-to-source compilation with various source-level optimization support for many languages including Fortran and C/C++ with OpenMP. It bears similarity to XLFLANG and even more so to CLANG in that it implements various transformations and optimizations including OpenMP 3.0 support by manipulating its internal AST. XLFLANG focuses mainly on efficiently translating a high-level IR to another high-level IR while minimizing overheads from language differences. Also XLFLANG and CLANG/LLVM combined provides up-to-date OpenMP 4.5 support, compared to OpenMP 3.0 support in Rose.

More recently, an open-source development project for Fortran front-end for LLVM, FLANG [13], has launched. Since the project is still ongoing, there is not enough information to compare with XLFLANG.

7 Conclusions

Rapidly evolving parallel architectures and programming models led to the reduced cycle of compiler and runtime development. We believe our approach to XLFLANG is in line with such trend, minimizing redundant effort while augmenting existing compiler infrastructure. Bridging one IR to another IR with completely different assumptions and backgrounds was not without challenges, but our work showed that it could be done with proper understanding of the base languages, Fortran and C/C++, and OpenMP requirements. Our future work includes adding AST-level optimizations to XLFLANG, interfacing with CLANG/LLVM to convey alias information from Fortran, and extending its support to more recent Fortran family.

Acknowledgement. This work was supported in part by the United States Departmnet of Energy CORAL program (contract B604142). We thank the IBM XL compiler team for their help and support for this work.

References

1. F2cpp: a python script to convert fortran 77 to C++ code. http://sourceforge.net/projects/f2cpp
2. OpenMP homepage. http://www.openmp.org/
3. Antao, S.F., Bataev, A., Jacob, A.C., Bercea, G.T., Eichenberger, A.E., Rokos, G., Martineau, M., Jin, T., Ozen, G., Sura, Z., Chen, T., Sung, H., Bertolli, C., O'Brien, K.: Offloading support for OpenMP in Clang and LLVM. In: LLVM-HPC 2016 (2016)
4. Barrowes, B.: F2matlab. http://engineering.dartmouth.edu/~d30574x/consulting/consultingIndex.html
5. Chen, T., Sura, Z., Sung, H.: Automatic Copying of Pointer-Based Data Structures (2017)
6. IBM Corporations: Xl fortran for linux. http://www-03.ibm.com/software/products/en/xlfortran-linux
7. Dietrich, R., Juckeland, G., Wolfe, M.: Open ACC programs examined: a performance analysis approach. In: 2015 44th International Conference on Parallel Processing, pp. 310–319, September 2015
8. Feldman, S.I., Gay, D.M., Maimone, M.W., Schryer, N.L.: A Fortran to C converter (1990)
9. Grosse-Kunstleve, R.W., Terwilliger, T.C., Sauter, N.K., Adams, P.D.: Automatic Fortran to C++ conversion with fable. Source Code for Biology and Medicine (2012)
10. JamesBeyer, J.L.: Targeting GPUs with OpenMP 4.5 device directives. http://on-demand.gputechconf.com/gtc/2016/presentation/s6510-jeff-larkin-targeting-gpus-openmp.pdf
11. Liao, C., Quinlan, D.J., Panas, T., Supinski, B.R.: A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 15–28. Springer, Heidelberg (2010). doi:10.1007/978-3-642-13217-9_2
12. Peterson, P.: F2py: Fortran to Python interface generator. http://cens.ioc.ee/projects/f2py2e
13. L.F. Team. Flang. https://github.com/llvm-flang
14. T.C. Team. Clang 5 documentation: Libtooling. https://clang.llvm.org/docs/LibTooling.html