2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)

Clacc: Translating OpenACC to OpenMP in Clang

Joel E. Denny, Seyong Lee, Jeffrey S. Vetter
Oak Ridge National Laboratory
Email: {dennyje, lees2, vetter}@ornl.gov

***Abstract*—OpenACC was launched in 2010 as a portable programming model for heterogeneous accelerators. Although various implementations already exist, no extensible, open-source, production-quality compiler support is available to the community. This deficiency poses a serious risk for HPC application developers targeting GPUs and other accelerators, and it limits experimentation and progress for the OpenACC specification. To address this deficiency, Clacc is a recent effort funded by the US Exascale Computing Project to develop production OpenACC compiler support for Clang and LLVM. A key feature of the Clacc design is to translate OpenACC to OpenMP to build on Clang's existing OpenMP compiler and runtime support. In this paper, we describe the Clacc goals and design. We also describe the challenges that we have encountered so far in our prototyping efforts, and we present some early performance results.**

***Index Terms*—OpenACC, OpenMP, LLVM, multicore, GPU, accelerators, source-to-source translation, compiler**

I. Introduction

Heterogeneous and manycore processors (e.g., multicore CPUs, GPUs, FPGAs) are becoming de facto architectures for current HPC and future exascale platforms [1]–[3]. These architectures are drastically diverse in functionality, performance, programmability, and scalability, significantly increasing complexity that HPC application developers face as they attempt to fully utilize available hardware. Moreover, users plan to run on many platforms during an application's lifetime, so this diversity creates an increasingly critical need for performance portability of applications and related software.

The Exascale Computing Project (ECP) and the broader community are actively exploring strategies to provide this portability and performance. These strategies include libraries, domain-specific languages, OpenCL, and directive-based compiler extensions like OpenMP and OpenACC. Over the last decade, most HPC users at the DOE have used either CUDA or OpenACC for heterogeneous computing. CUDA has been very successful for NVIDIA GPUs, but it's proprietary and available on limited architectures. OpenMP has only recently provided limited support for heterogeneous systems as historically it has focused on shared-memory multi-core programming. OpenCL has varying levels of support across platforms and is often verbose, and thus it must be tuned for each platform.

OpenACC was launched in 2010 as a portable, directive-driven, programming model for heterogeneous accelerators [4]. Championed by organizations like NVIDIA, PGI, and ORNL, OpenACC has evolved into one of the most widely used portable programming models for accelerators on HPC systems today. Because OpenACC is specified as a more *descriptive* language [5] while OpenMP is more *prescriptive*, the compiler has more freedom in how it implements OpenACC directives, placing more optimization power in the compiler's hands and depending less on the application developer for performance tuning. The current version of the OpenACC specification is 2.6, released in November 2017.

A variety of OpenACC implementations exist, from production proprietary compilers to research compilers, including PGI [6], OpenARC [7]–[9], GCC [10], and Sunway OpenACC from Wuxi [11], [12], providing various support for offloading to NVIDIA GPU, AMD GCN, multicore CPU, Intel Xeon Phi, and FPGA. However, in this time of extreme heterogeneity in HPC architectures, we believe it's critical to have an open-source OpenACC implementation to facilitate broad use, to enable researchers to experiment on a broad range of architectures, and to provide a smooth transition path from research implementations into production deployments. Currently, the only production open-source OpenACC compiler cited by the OpenACC website is GCC [13]. GCC's support is relatively new and so lags behind commercial compilers, such as PGI, to provide production support for the latest OpenACC specs [14]. Also, GCC has a reputation for being challenging to extend and, especially within the DOE, is losing favor to Clang and LLVM for new compiler research and development efforts.

A. Clacc Objectives

Clacc is a recent effort to develop an open-source, production OpenACC compiler ecosystem that is easily extensible and utilizes the latest advances in compiler technology. Such an ecosystem is critical to successful acceleration of applications using modern HPC hardware. Clacc's objectives are:

1) Develop production, standard-conforming OpenACC compiler/runtime support by extending Clang/LLVM.
2) As part of the compiler design, leverage the Clang ecosystem to enable research and development of source-level OpenACC tools, such as pretty printers, analyzers, lint tools, and debugger and editor extensions.
3) As the work matures, contribute OpenACC support to upstream Clang and LLVM so that it can be used by the broader HPC and parallel programming communities.
4) Throughout development, actively contribute upstream all Clang/LLVM improvements, including OpenMP improvements, that are mutually beneficial to OpenACC support and to the broader Clang/LLVM ecosystem.
5) Throughout development, actively contribute improvements to the OpenACC specification.

A key feature of the Clacc design is to translate OpenACC to OpenMP to build on Clang's existing OpenMP compiler and

DOI 10.1109/LLVM-HPC.2018.00006

runtime support. Because OpenACC is more descriptive while OpenMP is more prescriptive, this translation is effectively a lowering of the representation and thus fits the traditional ordering of compiler phases. We primarily intend this design as a pragmatic choice for implementing traditional compiler support for OpenACC. Thus, like other intermediate representations, the generated OpenMP and related diagnostics are not normally exposed to the compiler user.

Even so, this design also creates the potential for several interesting non-traditional user-level features. For example, this design offers a path to reuse existing OpenMP debugging and development tools, such as ARCHER [15], for the sake of OpenACC. This design should give rise to semantics and compiler support for the combination of OpenACC and OpenMP in the same application source. As a lower representation, OpenMP could serve as a debugging and analysis aid as compiler developers and application authors investigate the optimization decisions made by the OpenACC compiler.

Perhaps the most obvious user-level feature that Clacc's design enables is source-to-source translation. This feature is especially important for the following reason. While we believe that OpenACC's current momentum as the go-to directive-based language for accelerators will continue into the foreseeable future, it has been our experience that some potential OpenACC adopters hesitate over concerns that OpenACC will soon be supplanted by OpenMP features. As a tool capable of automatically porting OpenACC applications to OpenMP, Clacc could neutralize such concerns, encourage adoption of OpenACC, and thus advance utilization of acceleration hardware in HPC applications.

B. Contributions

Clacc is a work in progress, and it remains in active development. We are currently prototyping OpenACC support and evolving our design as issues arise. The contributions of this paper are as follows:

1) We explain the rationale for Clacc's design as well as the current state of our prototype.
2) We describe our extension of Clang's `TreeTransform` facility for transforming OpenACC to OpenMP without breaking Clang's immutable AST design.
3) We describe a prescriptive mapping from OpenACC to OpenMP for directives and clauses Clacc so far supports.
4) We discuss how Clacc's design may evolve in the future to utilize LLVM IR-level analyses, potentially taking advantage of LLVM IR parallel extensions that are currently being discussed in the community.
5) We present early performance results for our prototype.

II. Methodology

In this section, we present the current design of our Clacc prototype. As discussed in §I, extending Clang to translate OpenACC to OpenMP appears to be the most pragmatic approach for extending the LLVM ecosystem to support OpenACC in C and C++, and this approach also enables a number of helpful non-traditional user-level features. However, Clang was not originally designed to support AST transformations or source-to-source translation. Thus, to ensure the success of the Clacc project, it is important that we carefully weigh the potential ramifications of each major design decision.

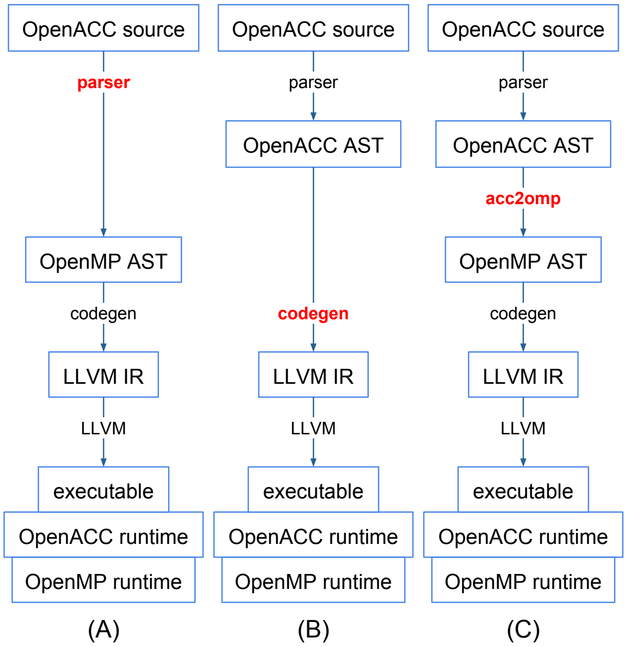


Fig. 1. Clacc Design Alternatives

A. High-Level Design

In this section, we evaluate three high-level design alternatives we have considered for the Clacc compiler. We have previously posted this portion of the design and discussed it on the Clang developer mailing list, but we feel that it is important to recapture it here to provide context for the rest of the paper.

Fig. 1 depicts our three high-level design alternatives. In each design alternative, red indicates the compiler phase that effectively translates OpenACC to OpenMP. The components of this diagram are as follows:

- **OpenACC source** is C/C++ application source code containing OpenACC constructs.
- **OpenACC AST** is a Clang AST in which OpenACC constructs are represented by OpenACC node types, which don't already exist in Clang's upstream implementation.
- **OpenMP AST** is a Clang AST in which OpenACC constructs have been lowered to OpenMP constructs represented by OpenMP node types, which do already exist in Clang's upstream implementation.
- **LLVM IR** is the usual LLVM intermediate representation generated by Clang.
- **Parser** is the existing Clang parser and semantic analyzer extended for OpenACC. Under design A, this component parses OpenACC directly into an OpenMP AST.
- **acc2omp** is design C's new Clang component that transforms OpenACC to OpenMP entirely at the AST level.

- **Codegen** is the existing Clang backend that lowers a Clang AST to LLVM IR. Design B extends this component to lower OpenACC node types into LLVM IR with runtime calls.
- **executable** is the final application executable.
- **OpenACC runtime** is built on top of LLVM's existing **OpenMP runtime** with extensions for OpenACC's runtime environment variables, library API, etc.

There are several design features to consider when choosing among these design alternatives:

1) **OpenACC AST as an artifact**: Because they create OpenACC AST nodes, designs B and C best facilitate the creation of additional OpenACC source-level tools, as described for Clacc objective 2 in §I. Some of these tools, such as pretty printing, are available immediately or as minor extensions of tools that already exist in Clang's ecosystem.
2) **OpenMP AST/source as an artifact**: Because they create OpenMP AST nodes, designs A and C best facilitate the development of the various non-traditional user-level features discussed in §I, such as automated porting of OpenACC applications to OpenMP, and reuse of existing OpenMP tools. With design B instead, the only OpenMP representation is low-level LLVM IR plus runtime calls, from which it would be significantly more difficult or impossible to implement such features.
3) **OpenMP AST for mapping implementation**: Designs A and C also make it easier for the compiler developer to reason about and implement mappings from OpenACC to OpenMP. That is, because OpenACC and OpenMP syntax is so similar, implementing the translation at the level of a syntactic representation is easier than translating to LLVM IR.
4) **OpenMP AST for codegen**: Designs A and C simplify the compiler implementation by enabling reuse of Clang's existing OpenMP support for codegen. In contrast, design B requires either significant extensions to Clang codegen to support OpenACC nodes, or a signficant refactoring of the OpenMP codegen to make it reusable for OpenACC.
5) **Full OpenACC AST for mapping**: Designs B and C potentially enable the compiler to analyze the entire source, as opposed to just the OpenACC construct currently being parsed, while choosing the mapping to OpenMP. It is not clear if this feature will prove useful, but it might enable more optimizations and compiler research opportunities.
6) **No OpenACC node classes**: Design A simplifies the implementation by eliminating the need to implement many OpenACC node classes. While we have so far found that implementing these classes is mostly mechanical, it does take a non-trivial amount of time as the code for an AST class is divided among many Clang facilities, such as the AST printer, dumper, reader, writer, visitor, etc.
7) **No OpenMP mapping**: Design B does not require OpenACC to be mapped to OpenMP. That is, it is conceivable that, for some OpenACC constructs, there will prove to be no OpenMP syntax to capture the semantics we wish to implement. It is also conceivable that we might one day want to represent some OpenACC constructs directly as extensions to LLVM IR, where some OpenACC analyses or optimizations might be more feasible to implement. This possibility dovetails with recent discussions in the LLVM community about developing LLVM IR extensions for various parallel programming models.

Due to features 4 and 6, design A is likely the fastest design to implement, at least at first while implementing simple OpenACC features and simple mappings to OpenMP. Even so, we have so far found no advantage that design A has but that design C doesn't have except feature 6, which we see as the least important of the above features in the long term.

The only advantage we have found that design B has but that design C does not have is feature 7. It should be possible to choose design C as the default but, for certain OpenACC constructs or scenarios where feature 7 proves important, if any, incorporate design B. In other words, if we decide not to map a particular OpenACC construct to any OpenMP construct, acc2omp would leave it alone, and we would extend codegen to handle it directly.

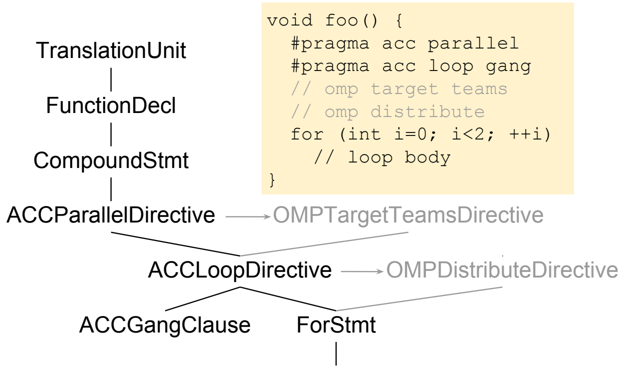
For the above reasons, and because design C offers the cleanest separation of concerns, we have chosen design C with the possibility of incorporating design B where it proves useful. So far in our prototype, we have not needed to incorporate design B.

B. acc2omp Background

As described in the previous section, our Clacc design introduces a new acc2omp component within Clang. A key issue in designing acc2omp is that Clang ASTs are designed to be immutable once constructed. This issue might at first seem to make acc2omp impossible to implement, but it does not. Here are a few AST transformation techniques that are already in use in upstream Clang:

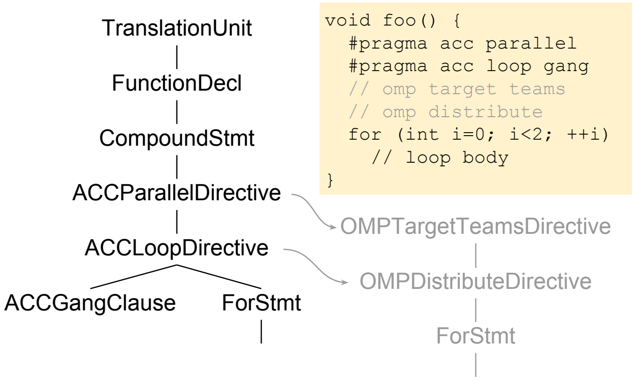
1) Clang has a `Rewrite` facility for annotating an AST with textual modifications and printing the resulting source code, which can then be reparsed as new source code into a new AST.
2) The initial construction of a Clang AST involves attaching new nodes to existing nodes, so modifying an AST by extending it is permitted by design.
3) The `TreeTransform` facility in Clang is currently used to transform C++ templates for the sake of instantiating them. It is a special case of technique 2 as follows. When the parser reaches a template instantiation, `TreeTransform` builds a transformed copy of the AST subtree that represents the template, and it inserts that copy into the syntactic context of the template instantiation, thus extending the AST.

One nice property of `TreeTransform` is that it is designed to be extensible: it is a class template employing the curiously



```
void foo() {
  #pragma acc parallel
  #pragma acc loop gang
  // omp target teams
  // omp distribute
  for (int i=0; i<2; ++i)
    // loop body
}
```

Fig. 2. AST with Hidden OpenMP Parent



```
void foo() {
  #pragma acc parallel
  #pragma acc loop gang
  // omp target teams
  // omp distribute
  for (int i=0; i<2; ++i)
    // loop body
}
```

Fig. 3. AST with Hidden OpenMP Subtree

recurring template pattern (CRTP) for static polymorphism. However, there are also a few caveats, which we now consider.

1) `TreeTransform` Caveat 1: Transitory semantic data: To build new nodes, `TreeTransform` runs many of the same semantic actions that the parser normally runs. Those semantic actions require the transitory semantic data that has been stored in Clang's `Sema` object by the time the parser reaches the syntactic context where new nodes are to be inserted, but the parser gradually discards some of that semantic metadata as the parser progresses to other syntactic contexts. Thus, `TreeTransform` cannot be run on arbitrary nodes in the AST at arbitrary times. For example, to run `TreeTransform` on arbitrary nodes in a translation unit after the parsing of that translation unit has completed, it might be necessary to transform the translation unit's entire AST in order to rebuild all of the necessary transitory semantic metadata.

2) `TreeTransform` Caveat 2: Permanent semantic data: Currently, parsing a C++ template permanently associates semantic data with that template's AST subtree in a way that's compatible with later runs of `TreeTransform` for instantiations of that template. However, there's no guarantee that semantic data that is reasonable for C++ template instantiation will be compatible with any arbitrary extension of `TreeTransform`. For example, we have noticed that, if we write a simple `TreeTransform` extension that merely duplicates an OpenMP region immediately after that region's node is constructed, the default `TreeTransform` implementation does not update the declaration contexts for variable declarations that are local to the duplicate region, so those duplicate variables appear to be declared in the original region, resulting in spurious compiler diagnostics.

3) `TreeTransform` Caveat 3: Unknown limitations: Because `TreeTransform` is designed primarily for C++ template instantiation, it is not clear at this point what other limitations an attempt to extend it for Clacc might expose.

C. acc2omp Design

We consider the following design alternatives for acc2omp:

1) Alternatives for *how* transformations are performed:
    a) **Annotate** each OpenACC node with OpenMP as text using Clang's `Rewrite` facility, print the entire AST, and parse that to construct an OpenMP AST.
    b) **Replace** each OpenACC node with an OpenMP node.
    c) **Add a second hidden parent**, an OpenMP node, for the children of each OpenACC node. For example, Fig. 2 depicts how the OpenACC node for an `acc loop` directive remains in the main AST but holds a pointer to a hidden `omp distribute` directive node that shares a child. The OpenACC node delegates codegen to the OpenMP node as necessary, but the OpenMP node is skipped by any traversals related to analysis of the original source.
    d) **Add a hidden subtree**, rooted at an OpenMP node, for each OpenACC node. This alternative is the same as the previous alternative except that the OpenACC node and OpenMP node do not share children. Instead, the entire subtree is transformed using `TreeTransform`. For example, Fig. 3 depicts how an `acc loop` directive node holds a pointer to the hidden subtree for its `omp distribute` directive.
    e) **Add a new translation unit** with OpenMP nodes instead of OpenACC nodes. This approach could also use `TreeTransform`.
2) Alternatives for *when* transformations are performed:
    a) Immediately after **each OpenACC node** is constructed. This approach is nonsensical for 1e above.
    b) At the end of **each translation unit** or all translation units. For 1d above, this approach is likely infeasible due to the `TreeTransform` caveat mentioned in §II-B1. If using `TreeTransform`, 1e seems necessary to overcome that caveat.

We reject the first two alternatives under 1 for the following reasons. Alternative 1a is surely bad for compilation performance as it requires serializing and reparsing the source AST. While alternative 1b seems to directly contradict the Clang AST design, we have spoken to developers who have successfully employed this strategy in forks of older versions of Clang. Regardless, because it blatantly contradicts the Clang design, it might easily break as Clang evolves.

In our initial Clacc prototype, we attempted 1c plus 2a as that alternative seemed sufficient for simple mappings from OpenACC to OpenMP, it seemed to avoid the need for something more complex like `TreeTransform`, and it seemed easiest to extend to 2b in case we later determine that full translation-unit visibility is sometimes required for a transformation. However, we encountered several issues with this alternative. The most obvious problem is that, because of the shared children, it doesn't permit changes to an OpenACC directive's associated code block when translating to OpenMP, but we have found that such changes are necessary, as we describe in §II-G. Nested OpenACC directives pose a related problem: the outermost OpenACC node logically has different children than its hidden OpenMP node because the nested directive is OpenACC in the former case and OpenMP in the latter case. It's possible to maintain the same children and attach a hidden OpenMP node to each of the outer and inner OpenACC nodes, but then AST traversals for OpenMP, most notably OpenMP codegen, must be modified to handle OpenACC nodes within OpenMP subtrees. For example, in Fig. 2, the child of the `omp target teams` directive node is an `acc loop` directive node.

A more subtle issue we encountered with alternative 1c is that it means that an OpenACC node's children must be compatible with the OpenMP node type. For example, when trying to translate an `acc parallel` construct to an `omp target teams` construct using this alternative, Clang's OpenMP implementation required us to insert child nodes representing two captured regions, one for each of `omp target` and `omp teams`, as parents of the associated code block. Because Clang's AST is immutable, the subtree for the associated code block cannot be modified later, and thus the choice of how to map a particular OpenACC directive to OpenMP must be made immediately in order to construct the subtree correctly. The choice cannot be deferred for the sake of, for example, alternative 2b. This example also points out a more general problem with alternative 1c: understanding how to construct child nodes requires understanding and sometimes replicating portions of the OpenMP semantic analysis. In contrast, `TreeTransform` offers a nice encapsulation for reusing semantic analysis implementations during transformation.

Our current Clacc prototype employs alternative 1d plus 2a. This design seems:

- **Most compatible**: Unlike 1b, 1d doesn't attempt to violate Clang AST immutability.
- **Most efficient**: Unlike 1a or 1e, 1d doesn't require rebuilding the entire AST.
- **Most flexible**: 1d avoids the many issues cited above for 1c. 1d plus 2a is like 1e plus 2b but on a smaller scale, so the former ought to be extensible to the latter if we later determine that full translation-unit visibility is sometimes required for a transformation.

We avoid the `TreeTransform` caveat from §II-B1 by combining 1d only with 2a. We have overcome the `TreeTransform` caveat from §II-B2 by overriding specific `TreeTransform` functionality in order to transform semantic data that `TreeTransform` normally leaves unmodified. We anticipate that the extensible design of `TreeTransform` will enable us to overcome `TreeTransform` issues we encounter in the future, as discussed in §II-B3. For extensions that seem general to rewriting directives rather than specific to translating OpenACC to OpenMP, we are constructing a separate class that we hope will prove reusable in projects other than Clacc.

D. OpenMP Implementation Reuse

In our Clacc prototype, we have found that the design discussed so far maximizes Clacc's reuse of the existing OpenMP implementation in Clang and LLVM. First, as discussed in §II-A, full reuse of OpenMP codegen is enabled by constructing an OpenMP AST. Second, because of our choice of alternative 1d from §II-C for acc2omp, `TreeTransform` enables reuse of the OpenMP semantic analysis implementation without breaking encapsulation. Finally, as discussed in §II-A, our OpenACC runtime will be built on the OpenMP runtime plus extensions for OpenACC's run-time environment variables, library API, etc. However, our prototype currently uses the OpenMP runtime directly and doesn't yet provide support for runtime-related OpenACC features, so we don't discuss runtime design further in this paper.

E. Traversing OpenACC vs. OpenMP AST

As described in §II-C, the manner in which we use TreeTransform maintains both the OpenACC nodes and their corresponding OpenMP nodes in the AST at the same time. For this reason, Clang tools and built-in source-level functionality need some way to choose which set of nodes to examine. In this section, we describe our solutions for two existing Clang facilities, AST printing and dumping, and we consider possible mechanisms for other AST traversals.

1) Printing: `-ast-print` is an existing Clang command-line option for printing the original pre-processed source code from the Clang AST. While this feature was designed for debugging and not for faithful printing of the AST, we have successfully contributed upstream a number of fixes to improve the fidelity of its output. In all of our Clacc tests so far, we can successfully compile and run the printed source without behavioral changes.

As we discussed in §I, source-to-source translation is an important user-level feature that Clacc's design enables. As a first prototype of this feature, we are extending `-ast-print`. However, the output of `-ast-print` always corresponds to the original source and never a lowered version of it. Clacc maintains that behavior, so `-ast-print` prints the OpenACC source not the OpenMP source. Thus, in Clacc, our extended `-ast-print` functionality is accessed via a new option, `-fopenacc-print`, which takes any of the following values:

1) `acc`: OpenACC subtrees are printed, and OpenMP subtrees to which they were translated are ignored. In this case, the only difference from `-ast-print` is that `-ast-print` typically must be combined

with several other command-line options to, for example, enable OpenACC compilation. That is, -fopenacc-print=acc is more convenient.

2) omp: OpenMP subtrees are printed, and the OpenACC subtrees from which they were translated are ignored.
3) acc-omp: OpenACC subtrees are printed, and the OpenMP subtrees to which they were translated are printed in neighboring comments.
4) omp-acc: OpenMP subtrees are printed, and the OpenACC subtrees from which they were translated are printed in neighboring comments.

In the last two cases, Clacc will avoid duplicating the code block associated with a directive if that code block prints identically in both the OpenACC and OpenMP versions. The output then looks similar to the code passage in Fig. 3.

In the future, we plan to investigate other alternatives on which to base -fopenacc-print, such as the clang-format tool, which offers more faithful printing, better formatting, and an unpreprocessed version of the source.

2) Dumping: -ast-dump is an existing Clang command-line option for printing a textual representation of the AST structure, including parent-child relationships, source location information, and computed types. This feature is clearly designed for debugging ASTs and is not for normal Clang users. Nevertheless, it is very useful for Clang developers and must be supported by Clacc. For each OpenACC AST node, we have extended this feature to always produce a full representation of that node's subtree including, as a specially marked child node, the OpenMP subtree to which it translates.

3) Other Traversals: AST traversals are typically based on Clang's RecursiveASTVisitor facility. Like most -ast-print users, most AST traversal developers and users likely expect for traversals to visit an AST representing the original source code only. Because the OpenMP node to which an OpenACC node is translated is not recorded as a normal child of the OpenACC node, RecursiveASTVisitor visits the OpenACC node but skips its OpenMP node. However, while visiting an OpenACC node, a visitor can be written to call the node's getOMPNode member function to access the OpenMP node, possibly for a recursive visitation. For example, we use that member function for implementing -fopenacc-print and -ast-dump.

If a developer wishes to use an existing Clang-based tool to operate on only the OpenMP AST to which Clacc translates an OpenACC AST, but if he doesn't wish to rewrite the tool to call getOMPNode on all OpenACC nodes, he can use -fopenacc-print=omp and then run Clang again to parse the output. However, it might be more efficient and convenient to have a mechanism that adjusts all AST traversals to skip OpenACC nodes and visit only their OpenMP nodes instead. In that way, the AST would appear to a tool as if it were constructed from parsing the generated OpenMP source. This mechanism might be activated by a new command-line option, such as -fopenacc-ast=omp. Some AST traversals, most notably codegen, would ignore such a mechanism. That is, under our chosen design C from §II-A, the implementation of codegen for an OpenACC node would delegate to the corresponding OpenMP node regardless of this mechanism. If we find cases where we need to incorporate design B, codegen would always operate directly on the OpenACC node regardless of this mechanism. More investigation is necessary to determine if such a mechanism is worthwhile in practice.

F. Development Strategy

Like any software development project, Clacc cannot offer support for all desired functionality at its inception. Instead, we are carefully studying each OpenACC feature, determining a correct mapping to OpenMP, implementing it in Clang, extending Clang's test suite with thorough coverage, and contributing relevant improvements to the OpenACC specification as well as to the Clang and LLVM upstream infrastructure. In order to set realistic milestones, we have started with the most fundamental and commonly used features, as indicated by their prevalence in existing OpenACC applications and benchmarks, and we are working our way to more complex and more rarely used features. We now discuss a few high-level limitations that we have imposed initially to facilitate this process. We will of course eliminate those limitations as Clacc matures.

1) C vs. C++: Clacc currently supports OpenACC directives in C but not yet C++. C++ is a vastly more complex language than C, so initially limiting our work to C facilitates faster progress toward full OpenACC support. Nevertheless, C++ support is an important objective for Clacc. Not only is C++ a critical base language supported by Clang, but C++ continues to see growing usage within HPC applications.

2) Shared-Memory Multicore vs. GPU: Clacc currently supports OpenACC offloading to shared-memory multicore CPUs but not yet GPUs. The primary reason is that our Clacc work has so far focused on compute constructs, loop constructs, and data-sharing features but does not yet support host-device data-transfer features. When the offloading target is shared-memory multicore CPUs, OpenACC compilers can mostly ignore the use of such data-transfer features [16]. Indeed, we take this approach for Clacc when compiling the benchmarks we evaluate in §III. Nevertheless, we expect to eliminate this limitation in the immediate future as we broaden the set of OpenACC features Clacc supports.

3) Prescriptive vs. Descriptive: A common and usually wise strategy when implementing support for an existing language feature set is to focus first on behavioral correctness and then on performance. We are following this strategy for Clacc. For this reason, Clacc currently offers a prescriptive interpretation of OpenACC. That is, Clacc currently avoids any complex compiler analyses and thus employs a mostly one-to-one mapping of OpenACC directives to OpenMP directives. We present some of those mappings in §II-G.

No production-quality language support can endure this strategy indefinitely, and that is particularly true in the case of OpenACC. As we discussed in §I, OpenACC is specified as a descriptive language. That is, for many features, such as OpenACC's kernels directive, a production-quality OpenACC compiler is expected to perform complex analyses in order to understand data flow, loop nests, and other control flow in the application source and to determine efficient strategies for scheduling work on the offloading device. Clacc will thus need to perform those analyses to lower OpenACC to the more prescriptive language of OpenMP. In section §III, we investigate several OpenACC benchmarks as we begin to evaluate the performance gap between hardened commercial OpenACC compilers, in particular PGI, and Clacc's initially prescriptive OpenACC interpretation.

The latest OpenMP 5.0 draft, TR7, proposes a descriptive `loop` directive [17], which corresponds to OpenACC's `loop` directive with $imp|exp$ `independent`. Once Clang supports TR7's `loop`, Clacc can utilize it to facilitate a descriptive interpretation of OpenACC. However, we are not aware of TR7 features to which the following OpenACC features can be directly mapped: (1) the `loop` directive's `auto` clause, which requires the compiler to determine whether loop iterations are data-independent, and (2) the `kernels` directive, which requires the compiler to automatically split a region into kernels. Thus, even with TR7's `loop` directive, a fully featured production-quality OpenACC implementation would still require the kinds of compiler analyses described above.

G. Mapping OpenACC to OpenMP

In this section, we describe a mapping from OpenACC directives and clauses to OpenMP directives and clauses. This mapping represents a conservative choice intended to always achieve correct OpenACC behavior. As Clacc evolves to support a descriptive interpretation of OpenACC and the requisite compiler analyses, this mapping will represent the base choice from which Clacc will look for deviations to improve performance of the application, and this mapping will represent the fall back choice if Clacc fails to find better mappings. Under Clacc's current prescriptive interpretation of OpenACC, Clacc supports no such analyses and so effectively always falls back to this mapping.

1) Mapping Notation: For conciseness, we use the following notation when describing clauses and data attributes:

- pre labels a data attribute that is *predetermined* by the compiler (that is, cannot be overridden by an explicit clause) and is not specified by an explicit clause.
- imp labels a data attribute that is *implicitly determined* by the compiler (that is, can be overridden by an explicit clause) and is not specified by an explicit clause.
- exp labels a clause, possibly specifying a data attribute, that is explicitly specified in the source.
- not labels a clause that isn't explicitly specified.
- $L\ C \to L'\ C'$ specifies that clause or data attribute C under the condition identified by label L maps to clause or data attribute C' under the condition identified by label L', where a label is pre, imp, exp, or not.
- $L|L'\ C \to L''\ C'$ specifies both of the following mappings:
  - $L\ C \to L''\ C'$
  - $L\ 'C \to L''\ C'$
- Mappings for per-variable data attributes and clauses are per variable and per directive.
- Mappings for other clauses are per directive.
- Where arguments to clauses are not specified on either end of the mapping, the mapping maintains the arguments as they are even if the clause name or location changes.

One theme throughout Clacc's mapping is that Clacc does not rely on implicit or predetermined attributes of OpenMP except for cases where an explicit clause is not permitted. That is, Clacc tries to make the exact behavior it intends to produce as explicit as possible in the generated OpenMP for the sake of debugging. Thus, $\to exp$ appears frequently below.

Any directive or clause not mentioned in this mapping is not yet supported by Clacc.

2) Semantic Clarifications: While developing this mapping, we found we had to make assumptions about a number of aspects of OpenACC semantics in C that are not clear in the OpenACC 2.6 specification. In many cases, it was the related behavior of the Clang OpenMP implementation that brought the need for those assumptions to our attention. We describe some of those assumptions here. We are in the process of communicating clarifications to the OpenACC technical committee, and we will evolve this mapping if necessary based on those discussions.

- It is an error if a variable has more than one of exp `firstprivate`, exp `private`, or exp `reduction` on an OpenACC directive. These have contradictory specifications for initialization of the local copy of the variable and for storing data back to the original variable.
- While OpenACC does not define a `shared` clause, Clacc assigns the OpenMP imp `shared` semantics to any variable that is referenced within an OpenACC construct and declared outside it and for which OpenACC semantics do not specify `firstprivate`, `private`, or `reduction`.
- exp `firstprivate`, exp `private`, or exp `reduction` for a variable of incomplete type is an error. A local copy must be allocated in each of these case, but allocation is impossible for incomplete types.
- exp `private` or exp `reduction` for a `const` variable is an error. The local copy of a `const` private variable would remain uninitialized throughout its lifetime. A reduction assigns to both the original variable and a local copy after its initialization, but `const` prevents that.
- Given some variable v, rules to assign imp `shared(v)` or imp `firstprivate(v)` on an `acc parallel` directive are ignored if the following rule would then produce an imp `reduction` for v on that `acc parallel`.
- Given some variable v that is declared outside an `acc parallel`, if (1) neither exp firstprivate(v) nor exp private(v) on that `acc parallel`, and (2) exp reduction(o:v) on any contained gang-partitioned `acc loop`, then this `acc parallel` has imp reduction(o:v). If the first condition doesn't hold but the second does, then the reduction refers to the gang-private copy of v, so no gang-reduction is implied.

- It is an error if, on a particular OpenACC directive, there exist multiple $imp|exp$ `reduction` with different reduction operators for a single variable v.
- The arguments to `num_gangs`, `num_workers`, and `vector_length` must be positive integer expressions. The `vector_length` argument must also be a constant expression. These restrictions are inherited from the OpenMP clauses to which Clacc maps these clauses.
- For a sequential `acc loop` directive (see §II-G4), if the loop control variable is just assigned instead of declared in the init of the attached `for` loop, the loop control variable is imp `shared` instead of pre `private`. Otherwise, there's no way to tell an aggressive OpenACC compiler to leave such a loop as a normal sequential loop in C, where the variable would normally have `shared` semantics in that its final value is visible after the loop.
- For any `acc loop` directive, exp `reduction` is not permitted on a loop control variable regardless of its data sharing. However, if the loop control variable is declared instead of just assigned in the init of the attached `for` loop, any reference to the variable's name in the directive's clauses refers to a different variable, so this rule does not apply.

3) Parallel Directive: Clacc's current mapping of an `acc parallel` directive and its clauses to OpenMP is as follows:

- `acc parallel` → `omp target teams`
- imp `shared` → exp `shared`
- $imp|exp$ `firstprivate` → exp `firstprivate`
- exp `private` → exp `private`
- $imp|exp$ `reduction` → exp `reduction`
- exp `num_gangs` → exp `num_teams`
- If exp `num_workers` with a constant-expression argument, and if there is a contained worker-partitioned `acc loop`, then exp `num_workers` → wrap the `acc parallel` in a compound statement and declare a local `const` variable with the same type and value as the exp `num_workers` argument.
- Else, translation discards exp `num_workers`.
- Translation discards exp `vector_length`.

4) Loop Directive: Clacc does not yet support the `acc kernels` directive or an orphaned `acc loop` directive, so an `acc loop` must appear in an `acc parallel` directive.

Clacc treats an `acc loop` directive as *sequential* if either (1) exp `seq`, (2) exp `auto`, or (3) *not* `gang`, *not* `worker`, and *not* `vector`. The latter two cases depend on the OpenACC compiler to determine the best way to parallelize the loop. However, as described in §II-F3, Clacc does not yet support the necessary analyses and so depends on the application developer to prescribe the parallelization, so Clacc makes the conservative choice of a sequential loop instead. The third case would certainly be the more straightforward case to improve because OpenACC specifies that the loop iterations are then required to be data-independent. This is a case where a simple AST-level analysis could go a long way for existing OpenACC applications that expect a descriptive interpretation: Clacc could add whichever of `gang`, `worker`, or `vector` doesn't interfere with any explicit occurrences of these clauses on other enclosing or nested loops.

Clacc's current mapping of a sequential `acc loop` directive and its clauses to OpenMP is as follows:

- The `acc loop` directive and the following clauses or attributes are discarded during translation:
  - exp `seq`, exp `independent`, exp `auto`
  - exp `gang`, exp `worker`, exp `vector`
  - exp `collapse`
  - pre `private`, imp `shared`, exp `reduction`
- exp `private` → wrap loop in a compound statement and declare an uninitialized local copy of the variable.

If Clacc does not treat an `acc loop` directive as sequential based on the above conditions, then it treats it as *parallelized*. In that case, Clacc's current mapping of the `acc loop` directive and its clauses to OpenMP is as follows:

- `acc loop` → `omp`
- exp `gang` → `distribute`
- exp `worker` → `parallel for`
- If neither this nor any ancestor `acc loop` is gang-partitioned or worker-partitioned, then → `parallel for` and → exp `num_threads(1)`. We add `parallel for` for this case because OpenMP does not permit `omp simd` directly inside `omp target teams`.
- exp `vector` → `simd`
- The output `distribute`, `parallel for`, and `simd` OpenMP directive components are sorted in the above order before all clauses, including the above `num_threads(1)`, regardless of the input clause order.
- If exp `worker`, then exp `num_workers` from ancestor `acc parallel` → exp `num_threads` where the argument is either (1) the original exp `num_workers` argument if it is a constant expression or (2) otherwise an expression containing only a reference to the local `const` variable generated for that exp `num_workers`. On the ancestor `acc parallel` and on all OpenACC directives nested between it and this `acc loop`, Clacc leaves the OpenMP data sharing attribute for the local `const` variable for `num_workers` as implicit. Making explicit the data sharing for a generated `const` scalar doesn't seem worth the implementation effort.
- If exp `vector`, then exp `vector_length` from ancestor `acc parallel` → exp `simdlen`.
- `collapse` → `collapse`
- If exp `worker` or if this and every ancestor `acc loop` until the ancestor `acc parallel` is not gang-partitioned and not worker-partitioned, then imp `shared` → exp `shared`.
- Else, imp `shared` → imp `shared`. This case must be implicit because `omp distribute` or `omp simd` without `parallel for` (which we add for `worker` or to be able to nest `omp simd` directly within `omp target teams`) does not support a `shared` clause, so we must rely on OpenMP implicit data sharing rules then.

- pre `private` for a loop control variable that is declared in the init of the attached `for` loop → pre `private`. Mapping to exp `private` would be erroneous because it would refer to a variable from the enclosing scope.
- If exp `vector`, then $pre|exp$ `private` for a loop control variable that is just assigned instead of declared in the init of the attached `for` loop. For that variable, $pre|exp$ `private` → pre `linear`, as required by OpenMP for `omp simd`. Then, wrap the `acc loop` in a compound statement, and declare an uninitialized local copy of the loop control variable.
- In all other cases, $pre|exp$ `private` → exp `private`.
- If exp `worker` or exp `vector`, then exp `reduction` → exp `reduction`.
- Else, exp `reduction` is discarded here during translation. A gang reduction for a gang-private variable is useless and so is discarded during translation. Gang reductions for other variables are discarded here but addressed by the data sharing semantics we assumed for `acc parallel` in §II-G2.

H. Combined Directive

The only combined OpenACC directive Clacc supports so far is `acc parallel loop`, which is translated in two stages:

1) **Translate from combined directive to effective separate directives.** Clacc performs this stage during the parse while constructing the `acc parallel loop` directive's AST node. Clacc builds the effective AST subtree containing the `acc parallel` and `acc loop` directives, and then it records the AST subtree for the outermost of those directives, `acc parallel`, as a hidden subtree of the `acc parallel loop` node. The associated code block for the `acc parallel loop` node is recorded like a normal AST child for each of the `acc parallel loop` node and the `acc loop` node.
2) **Translate from effective separate directives to OpenMP directives.** This stage is performed using the `TreeTransform` facility just as it normally would be for the separate directives. That is, the `acc parallel loop` node delegates to the `acc parallel` node.

The relationship between the `acc parallel loop` node and the `acc parallel` node is similar to the relationship between any non-combined OpenACC directive's node and its OpenMP node. Moreover, it effectively replaces that relationship. That is, most AST traversals, including `-ast-print`, visit the `acc parallel loop` node and skip over the hidden subtree for its effective `acc parallel` directive because the `acc parallel loop` node without the `acc parallel` subtree represents the original source. The `-ast-dump` facility prints the `acc parallel` node as a specially marked child node, which prints its OpenMP node as a specially marked child node. For codegen to LLVM IR, the `acc parallel loop` node delegates to its effective `acc parallel` node, which delegates to its OpenMP node.

Because the second stage of translation above is delegated to the effective `acc parallel` directive, `acc parallel loop` does not require a mapping to OpenMP. However, it and its clauses do require a mapping to its effective directives for the sake of the first stage, as follows:

- `acc parallel loop` → `acc parallel`, whose associated code block is an `acc loop`, whose associated code block is the associated code block from the `acc parallel loop`.
- exp `private` → exp `private` on the effective `acc loop`.
- exp `reduction` → exp `reduction` on each of the effective `acc parallel` and `acc loop`.
- Each remaining explicit clause is permitted on only one of the separate OpenACC directives, and so it is mapped to that directive.
- Predetermined and implicit attributes do not require a mapping to the effective directives because there are none because semantic analysis computes them only on the effective directives.

The choice to map `reduction` to the effective `acc parallel` in addition to the effective `acc loop` doesn't follow the OpenACC 2.6 specification. However, strictly speaking, OpenACC 2.6 specifies reductions are only for scalars, which are imp `firstprivate`, so by default the reduced value from an `acc parallel loop` is not visible after the `acc parallel loop`. The OpenACC technical committee is working to address this and other confusing points in the specification of reductions, and Clacc's current mapping represents our attempt to match the intended behavior.

I. LLVM IR Analysis

As we described in §II-C, Clacc currently implements translation from OpenACC to OpenMP within the Clang frontend in order maximize reuse of the existing OpenMP implementation. As we described in §II-F3, as Clacc's OpenACC support evolves to production-quality, it will require sophisticated compiler analyses to perform this translation. However, the LLVM ecosystem is designed so that such analyses are best performed at the level of LLVM IR. While many source-level analysis tools have been built at the Clang AST level, the LLVM IR infrastructure for compiler analyses is designed for reuse across source languages and across target architectures, thus invites more attention from the developer community, and has become far more robust. One of the major research goals as Clacc matures is to investigate what elements of the translation from OpenACC to OpenMP under a descriptive interpretation of OpenACC can be effectively performed at the Clang AST level and what elements are best implemented at the LLVM IR level. For the sake of these latter elements, we are following ongoing efforts within the LLVM community to extend LLVM IR with better support for parallelism, which might eventually be integrated into Clang's OpenMP support as well.

Moving OpenACC-to-OpenMP translation to the LLVM IR level creates a challenge for some of the user-level features, such as source-to-source translation, that we described in §I. That is, this move implies design B from §II-A, which does not provide feature 2, an OpenMP AST/source artifact. First, it is important to clarify that the primary objective of the Clacc project, as described in §I, is to provide production-quality OpenACC support for Clang and LLVM. All other features we have described are lesser objectives. Nevertheless, in case such features prove important to the community and our sponsors but conflict with a technical need to move translation to the LLVM IR level, we will investigate the following potential solution. We imagine extending Clang codegen to encode specific compiler analysis queries into the LLVM IR it generates. After LLVM passes complete, the Clang compiler driver could pass the results of those queries back to the Clang frontend for the sake of AST-level translation to OpenMP. We anticipate that such queries could be encoded as a special instance of a general LLVM IR-level representation for source-level directives, such as the representation being developed for LLVM IR parallelism efforts.

III. Evaluation

We now evaluate our current Clacc prototype using several OpenACC benchmarks from the SPEC ACCEL 1.2[1] suite: 303.ostencil, 304.olbm, and 314.omriq, which are all written in C. Workloads we used in this evaluation are those included with the SPEC ACCEL distribution. Our results are not compliant SPEC results in part because, as we describe in this section, we modified the benchmark source code to experiment with various characteristics of the OpenACC implementations.

We compare our Clacc results for these benchmarks against results we obtained for the same benchmarks when compiling with the PGI Community Edition 18.4 [6], identified in the remainder of this section as pgcc. The stark contrast of Clacc's currently prescriptive interpretation of OpenACC against the powerful optimization capability of one the oldest and most advanced OpenACC compilers commercially available should provide insight into the challenges that lie ahead in evolving Clacc into a production-quality OpenACC compiler.

Our test platform runs Ubuntu 18.04 with an Intel Core i7-7700HQ 2.80GHz CPU (8 threads), 32 GB DRAM, and an NVIDIA GeForce GTX 1050. As discussed in §II-F2, Clacc's offloading support is currently limited to multicore, so we were able to compile for the GPU only with pgcc.

A. Testing Methodology

We compiled and ran all our selected benchmarks under a series of test configurations, which vary in terms of compiler, compiler options, offloading device, and modifications to benchmark sources. We now describe those configurations.

host clacc: We compiled each benchmark using `clacc -Ofast` without `-fopenacc`, so OpenACC compilation was disabled. Thus, this configuration provides a baseline for detecting when Clacc's offloading support does not improve performance relative to sequential execution. When compiling benchmark 314.omriq with this configuration, we found we had to also specify the command-line option `-DSPEC_NO_INLINE`, or compilation failed with a linking error for an inlined function.

host pgcc: We compiled each benchmark using `pgcc -fast -acc -ta=host`. Thus, this configuration provides a baseline for detecting when pgcc's offloading support does not improve performance relative to sequential execution.

multicore pgcc: This configuration is for side-by-side comparison with Clacc's multicore offloading performance. We compiled each benchmark using `pgcc -fast -acc -ta=multicore`.

multicore clacc: This configuration is the focus of this evaluation. We modified all benchmarks to remove all occurrences of `acc data`, `acc update`, `pcopyin`, `pcopyout`, and `pcopy`, which Clacc does not yet support and which are not actually useful because the target is shared-memory multicore, as discussed in §II-F2. We compiled each benchmark using `clacc -Ofast -fopenacc`.

multicore clacc+pgcc: This configuration is the same as "multicore clacc" except that we generated OpenMP source from Clacc and compiled it using `pgcc -fast -mp -ta=multicore` instead of using Clang's OpenMP support. Thus, this configuration helps us to isolate the effects of Clacc's prescriptive interpretation of OpenACC from the effects of Clang's OpenMP support.

multicore clacc gwv: This configuration tries to manually overcome negative effects of Clacc's prescriptive interpretation of OpenACC. It's the same as "multicore clacc" except it further modifies the benchmark sources to (1) add `worker` to every `acc loop` specifying only `gang`, (2) add `gang worker vector` to every `acc loop` that doesn't specify any of these clauses, and (3) add `private` clauses to those that don't specify all their private variables correctly. Change 2 should have an especially significant impact because, as described in §II-G4, before this change, Clacc translates these loops as sequential loops. The reason for the last change is that, even though failures to specify variables as private don't cause misbehavior in the previous configurations, we have found that changing the loop parallelization can alter a compilation strategy that otherwise manages to mask such mistakes.

multicore clacc+pgcc gwv: This configuration enables us to see if "multicore clacc gwv" performs differently when the generated OpenMP is compiled with `pgcc -fast -mp -ta=multicore` instead of Clang's OpenMP support.

gpu pgcc: This configuration examines whether the GPU can outperform multicore for our benchmarks, and so it gives us evidence of how soon Clacc development should focus on GPU support. We compiled each benchmark using `pgcc -fast -acc -ta=tesla:cc60`.

For all test configurations, we ran the benchmarks using SPEC's `runspec` script. In most cases, we accepted the default of three iterations per benchmark and ran the `test`, `train` and `ref` workloads included with SPEC ACCEL. The only exceptions were for a few configurations for 314.omriq, where the benchmark ran for many hours with the `ref` workload, so we chose to run only one iteration with the `ref` workload. As is usual for SPEC benchmarks, only the durations for the `ref`

[1] SPEC® and SPEC ACCEL® are registered trademarks of the Standard Performance Evaluation Corporation. For more information about SPEC ACCEL, see https://www.spec.org/accel/.

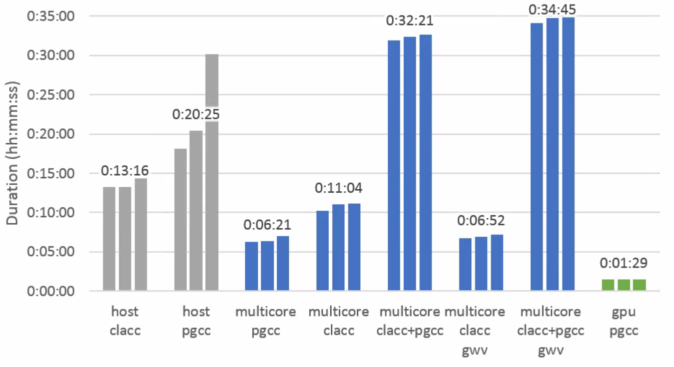


Fig. 4. 303.ostencil

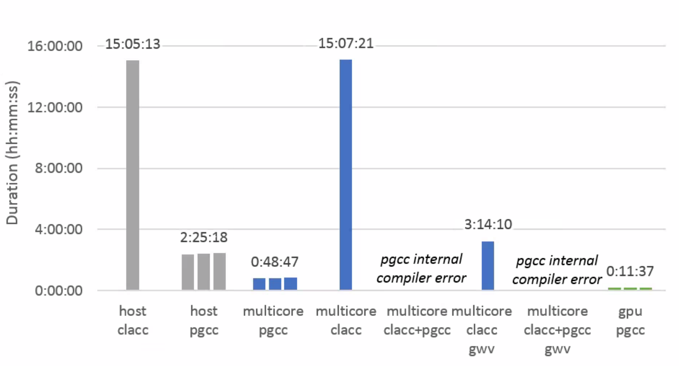


Fig. 6. 314.omriq

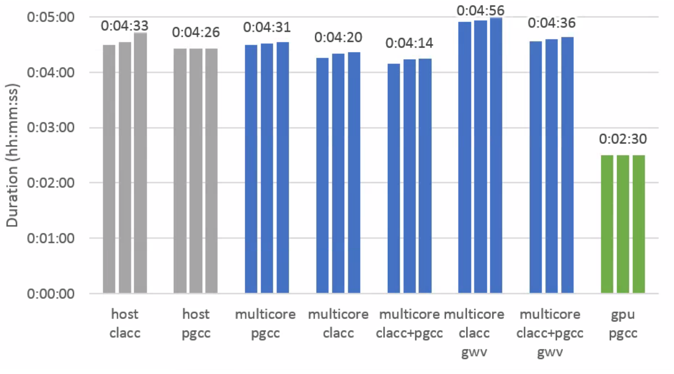


Fig. 5. 304.olbm

workloads are long enough to be interesting, and the `test` and `train` workloads are interesting for verifying correct behavior.

B. Results

Other than a few cases that failed compilation, discussed below, all benchmarks, all test configurations, all workloads, and all iterations passed verification of the benchmark output. Fig. 4, 5, and 6 present the duration of each iteration of each benchmark on the `ref` workload for each test configuration. As seen in these figures, the iterations within each group of three tend to have similar durations. However, in a few cases, we found that, if we ran the same experiment at another time, the result was sometimes another group of three durations that were similar to each other but noticeably different from the prior three. We expect the culprit is simply OS noise. Thus, for these results, we assume that small changes across test configurations are probably not meaningful.

303.ostencil is a thermodynamics benchmark solving partial differential equations (PDE) using a stencil code on a 3-D structured grid. pgcc offers a substantial performance improvement when moving offloading from host to multicore. Clacc offers a modest improvement for the same move, but it's small enough that it might be just OS noise. However, under the “multicore clacc gwv” configuration, Clacc is comparable with pgcc on multicore and improves significantly over Clacc's host performance. Interestingly, this configuration merely adds a single `worker` clause to a single gang loop that already has a nested vector loop. Another interesting point here is that attempting to compile Clacc's OpenMP output with pgcc gives significantly worse performance than either compiler does for host. On the other hand, pgcc offloading to GPU is far better than all other test configurations.

304.olbm is a computational fluid dynamics benchmark implementing the Lattice Boltzmann Method (LBM) and simulating incompressible fluids in 3D. pgcc offloading to GPU clearly outperforms the other test configurations for this benchmark. Otherwise, due to the aforementioned OS noise, we are not convinced the small differences in the various durations here are representative of important differences among the test configurations. It's no surprise that Clacc offloading to multicore fails to see a significant improvement over Clacc for host because 304.olbm specifies all `acc loop` directives without `gang`, `worker`, or `vector` clauses, so Clacc treats them as sequential loops. However, after we add those clauses, Clacc does not produce better performance, and pgcc, which applies the power of pgcc's descriptive interpretation of OpenACC, does no better either. We noticed that compilation for “multicore clacc gwv” warned of a failure to vectorize. We tried passing `-Minfo` to pgcc when offloading to multicore and found that then it too reported that, due to a data dependency, it was unable to vectorize each loop marked with `acc loop` in 304.olbm. However, when we offload to the GPU, pgcc reports successful vectorization.

314.omriq is a medical benchmark that performs MRI image reconstruction. This benchmark spends the vast majority of its time in a single loop that already has a gang outer loop and a vector inner loop. When compiling with Clacc and targeting host or multicore, the performance is poor enough that we didn't have time to run multiple iterations. Moreover, the performance is roughly the same, so the specified loop partitioning offers no measurable benefit over sequential execution. As for 303.ostencil, “multicore clacc gmv” adds a `worker` clause to the gang loop with a major positive impact on performance. However, performance is still worse than for pgcc compiling the benchmark sequentially. These results suggest that poor scalar optimizations could be to blame. As for the other benchmarks, moving to the GPU with pgcc offers dramatic performance improvements. For the Clacc plus pgcc compilation configurations, pgcc reported internal compiler errors, which are hard to debug because pgcc is closed source.

IV. Related Work

The home of the OpenACC specification and community is the OpenACC website [4]. Several text books on OpenACC exist to guide developers seeking to accelerate their applications [5], [16]. The OpenACC website lists a number of compilers and tools for OpenACC [13]. The only production open-source compiler cited is GCC [10], whose OpenACC support is primarily developed by Mentor Graphics. A recent blog from Mentor shows their attempts to improve GCC's OpenACC performance to match PGI's [18].

PGI's Michael Wolfe, the Technical Committee Chair for OpenACC, explains the signficance and complexity of translating OpenACC to OpenMP [19]. Sultana et al. prototyped a translator from OpenACC to OpenMP as an extension of the Eclipse C/C++ Development Tools [20]. Their primary objective is a simple, predictable source-to-source translation intended to be paired with "manual restructuring and performance tuning." Clacc's primary objective is production-quality OpenACC compiler support. Another difference is they map OpenACC's vector loops to `omp parallel for`, perhaps due to previously poor compiler support for `omp simd`.

V. Conclusions

We have presented Clacc, a recent project to develop production-quality OpenACC compiler support for Clang and LLVM. We have described Clacc's objectives, design, and the state of our prototype. A key design feature is to translate OpenACC to OpenMP to build on Clang's existing OpenMP compiler and runtime support. Clacc is in active development but currently only supports C, shared-memory multicore, and a prescriptive interpretation of OpenACC. We have presented an evaluation comparing the performance of several SPEC ACCEL benchmarks using our Clacc prototype vs. PGI. This evaluation confirms the need to extend with a more complex descriptive interpretation of OpenACC and with GPU support.

VI. Acknowledgments

We would like to thank the Clang and LLVM community for all their time and valuable input during mailing list discussions and code reviews as we have upstreamed our improvements to the Clang and LLVM infrastructure from the Clacc project. We would also like to thank Hal Finkel of Argonne National Laboratory for his technical feedback on an early version of the Clacc design that we posted to the Clang developer list.

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE), Office of Science. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

References

[1] S. Mittal and J. S. Vetter, "A survey of CPU-GPU heterogeneous computing techniques," *ACM Computing Surveys*, vol. 47, no. 4, pp. 1–35, 2015.

[2] B. Dally, "GPU computing to exascale and beyond," 2010.

[3] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," DARPA Information Processing Techniques Office, Tech. Rep., 2008.

[4] "OpenACC," [Online]. Available: https://www.openacc.org/.

[5] S. Chandrasekaran and G. Juckeland, Eds., *OpenACC for Programmers: Concepts and Strategies*. Addison-Wesley Professional, Sep 2017.

[6] "PGI," [Online]. Available: https://www.pgroup.com/.

[7] S. Lee and J. Vetter, "OpenARC: Open Accelerator Research Compiler for Directive-Based, Efficient Heterogeneous Computing," in *HPDC '14: Proceedings of the ACM Symposium on High-Performance Parallel and Distributed Computing, Short Paper*, June 2014.

[8] S. Lee, J. Kim, and J. S. Vetter, "OpenACC to FPGA: A Framework for Directive-Based High-Performance Reconfigurable Computing," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 544–554.

[9] A. Sabne, P. Sakdhnagool, S. Lee, and J. S. Vetter, "Understanding Portability of a High-Level Programming Model on Contemporary Heterogeneous Architectures," *IEEE Micro*, vol. 35, no. 4, pp. 48–58, July 2015.

[10] "GCC, the GNU Compiler Collection," [Online]. Available: https://gcc.gnu.org/.

[11] "Press Release: OpenACC Adoption Continues to Gain Momentum in 2016," [Online]. Available: https://www.openacc.org/news/press-release-openacc-adoption-continues-gain-momentum-2016, 2016.

[12] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, and et al., "The Sunway TaihuLight supercomputer: system and applications," *Science China Information Sciences*, vol. 59, p. 072001, Jun 2016.

[13] "OpenACC: Downloads & Tools," [Online]. Available: http://openacc.org/tools.

[14] K. Friedline, S. Chandrasekaran, M. G. Lopez, and O. Hernandez, "OpenACC 2.5 Validation Testsuite Targeting Multiple Architectures," in *High Performance Computing*. Cham: Springer International Publishing, 2017, pp. 557–575.

[15] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Mller, "ARCHER: Effectively Spotting Data Races in Large OpenMP Applications," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, p. 5362.

[16] R. Farber, Ed., *Parallel Programming with OpenACC*. Morgan Kaufmann, 2017.

[17] "OpenMP Technical Report 7: Version 5.0 Public Comment Draft," [Online]. Available: https://www.openmp.org/specifications/, July 2018.

[18] R. Allen, [Online]. Available: https://blogs.mentor.com/embedded/blog/2018/06/06/evaluating-the-performance-of-openacc-in-gcc/.

[19] M. Wolfe, "Compilers and More: OpenACC to OpenMP (and back again)," Jun 2016. [Online]. Available: https://www.hpcwire.com/2016/06/29/compilers-openacc-openmp-back/

[20] N. Sultana, A. Calvert, J. L. Overbey, and G. Arnold, "From OpenACC to OpenMP 4: Toward Automatic Translation," in *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*, ser. XSEDE16. New York, NY, USA: ACM, 2016, pp. 44:1–44:8.