Analysis of OpenMP 4.5 Offloading in Implementations: Correctness and Overhead

Jose Monsalve Diaz[a], Kyle Friedline[a], Swaroop Pophale[b], Oscar Hernandez[b], David E. Bernholdt[b], Sunita Chandrasekaran[a]

[a] *University of Delaware, 18 Amstel Avenue, Newark, Delaware 19716*
[b] *Oak Ridge National Laboratory, 1 Bethel Valley Road, Oak Ridge, Tennessee 37831*

Abstract

The OpenMP language features have been evolving to meet the rapid development in hardware platforms. This journal focuses on evaluating implementations of OpenMP 4.5 target offload features in compilers such as Clang, XL and GCC that are an integral part of the software harness on supercomputers and clusters. We use Summit (Top supercomputer in the world as of November 2018) as one of our experimental setup. Such an effort is particularly critical on such supercomputers as that is being widely used by application developers to run their scientific codes at scale. Our tests not only evaluate the OpenMP implementations but also expose ambiguities within the OpenMP 4.5 specification. We also assess the overhead of the different OpenMP runtimes in relationship to the different directives and clauses. This helps in assessing the interaction of different OpenMP directives independent of other application artifacts. We are aware that the implementations are constantly evolving and Summit is advertised as having only partial OpenMP 4.x support. This is a synergistic effort to help identify and fix bugs in features' implementations that are required by applications and prevent deployment delays later on. Going forward, we also plan to interact with standard benchmarking organizations like SPEC/HPG to donate our tests and mini-apps/kernels for potential inclusion in the next release versions of SPEC benchmark suite.

Keywords: OpenMP 4.5, Offloading, Overhead Measurement

1. Introduction

Top500 highlights mention that a total of 137 systems on the latest list use accelerator/co-processor technology up from 110 six months ago, of which sixty-four of them use NVIDIA Pascal, forty-six use NVIDIA Volta, thirteen systems that use NVIDIA Kepler, and four systems use Intel Xeon Phi co-processors [1]. The trend towards heterogeneous architecture (CPUs+Accelerator devices) only seems to be strengthening with more systems using different types of cores with each year. One key advantage of heterogeneous systems is the performance per watt contributed by accelerators in comparison to the traditional homogeneous CPU-based systems. These heterogeneous architectures offer tremendous potential with respect to performance gains, but attaining that potential requires scientific applications of thousands or even millions of lines of code to be migrated to support these architectures. Without appropriate support in performance-portable programming models that can exploit the rich feature sets of hardware resources, this already daunting task would be prohibitively difficult.

We need programming paradigms and tools to abstract the hardware differences across different platforms and provide reproducible and identical behavior for applications without burdening the scientific application developers to learn about the programming paradigms or intricacies of the hardware. One of the feasible and widely-adopted solutions to this challenge is using a directive-based programming model that allows programmers to insert hints into a given region of code for the compiler to automatically generate parallel code for the target system. Currently, the two popular directive-based programming models are OpenMP [2] and OpenACC [3]. Besides directive-based approaches, other techniques to program accelerators include CUDA [4], OpenCL [5], NVIDIA Thrust [6], and Kokkos [7].

OpenMP made a paradigm change to support heterogeneous systems and released specification versions 4.0, 4.5 and 5.0 in 2013, 2015, and 2018 respectively. New features include directives such as `simd` and `target`, additions to the `task` directive, such as the `taskloop`, `taskloop simd`, and `taskgroup` constructs and clauses for tasks such as `priority` and `depend`. Environment variables included hardware thread affinity description, affinity policies, default accelerator devices while runtime library routines included `omp_get_initial_device`, and other device memory routines. Technical reports have been augmenting recent releases of OpenMP with language features to manage memory on systems with heterogeneous memories, with

☆This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). 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).

*Jose Monsalve Diaz

Email addresses: `josem@udel.edu` (Jose Monsalve Diaz), `utimatu@udel.edu` (Kyle Friedline), `pophaless@ornl.gov` (Swaroop Pophale), `oscar@ornl.gov` (Oscar Hernandez), `bernholdtde@ornl.gov` (David E. Bernholdt), `schandra@udel.edu` (Sunita Chandrasekaran)

PREPRINT

features for task reductions, extensions to the target construct along with several clarifications and fixes. Some of these, included in OpenMP 5.0, include full support for accelerator devices, tool interfaces to allow third party tools development, better support for C++ data-types, enhancements to the loop construct, and memory allocation to support multilevel memory systems [8].

OpenMP 4.5 and above allows programmers to use the same standard to program CPU, SIMD units, and accelerators such as GPUs. Applications such as Pseudo-Spectral Direct Numerical Simulation-Combined Compact Difference (PSDNS-CCD3D) [9, 10], a computational fluid dynamics code on turbulent flow simulation rely on OpenMP 4.5 for on-node parallelism and run to scale on the Titan super-computer. Other applications that have used OpenMP 4.5 include Quick-silver, a Monte Carlo Transport code [11].

Compiler support for OpenMP 4.5 has increased in the recent years [12]. GCC versions 7.1 (May 2017) and up provide support for OpenMP 4.5 in C and C++, the most recent version GCC 9.1 has started integrating infrastructure for AMD accelerators. IBM XL (Dec 2016), in little endian Linux distributions, supports OpenMP 4.5 in C and C++ since V13.1.5, and some of the offloading features in Fortran since V15.1.15. Intel ICC 17.0, 18.0 and 19.0 compilers support OpenMP 4.5 for C, C++ and Fortran. For Cray systems the Cray Compiling Environment (CCE) 8.7 (April 2018) supports OpenMP 4.0 along with OpenMP 4.5 support for device constructs, Finally, LLVM Clang 3.8 released support for all non-offloading features of OpenMP 4.5, while version 7.0.0 introduces support for OpenMP 4.5 offloading to NVIDIA GPUs. A version of Clang that supports OpenMP 5.0 features as well as a Fortran front end for LLVM with OpenMP support, are currently under development.

As OpenMP's feature set continues to evolve, it is critical to ensure that their implementations conform to the specification. Maintaining consistency with the definition in the specification is a challenge as the description in the specification can often be interpreted in different ways. Due to such ambiguities, different compiler developers tend to interpret such descriptions differently and hence we find a particular feature implemented differently across different compilers. Sometimes these differences are quite subtle but trigger a productive discussion to fix the description in the specification.

For example, let us consider the usage and restrictions of the OpenMP's combined constructs. If one was using the `target teams` combined directive and wanted to map data to the device and then get a initialized copy of a data item from the master thread (firstprivatize), the `map` clause would only apply to the `target` directive and the `firstprivate` would only apply to the `teams` directive. Since the combined construct is semantically the same as writing each directives closely nested, one might expect for the code to execute properly. However, with a closer reading the `map` clause can only appear on a `target`, `target data`, `target enter data`, and `target exit data`. In each case, no data sharing attributes can be used on the directive. Hence, from reading the OpenMP specification alone it is not clear what combination of clauses are not permissible. Such ambiguities in specifications do not help the users (otherwise called as application developers) or the vendors (otherwise called as compiler developers). Our previous publications have captured more such discrepancies [13, 14].

While correctness and specification compliance in OpenMP implementations are critical, there are other aspects that users will consider for assessing quality of an implementation. Out of many, two important aspects are 1) quality of code generation, which has a direct impact in the application performance, and 2) quality of the OpenMP runtime system which should be transparent to the user and provide a low overhead to the final execution time. Although the former aspect is outside of the scope of this work, we have developed a methodology to evaluate the latter. Our approach is to measure the overhead introduced by the runtime when using the different OpenMP directives and clauses. By using NVIDIA's CUDA Profiling Tools Interface (CUPTI) [15], we are able to distinguish device runtime execution time and kernel execution time, from the OpenMP runtime execution time. We also evaluate the effect that number of threads and number of teams has over the measured overhead.

While runtime overhead may not necessarily have a direct impact in the overall performance of the application, studying it is important as it allows users (or applications) to understand the costs of using OpenMP offloading to acceleration devices. Additionally, these numbers directly relate to the complexity that the underlying runtime requires to support OpenMP offloading, as well as the runtime scaling across the threads and teams when mapped to the device. Furthermore, it provide users with additional insights to analyze the observed differences in the application behavior across implementations.

This journal is an extension of our recently published paper [16]. Of the four major features introduced in the OpenMP 4.5, offloading is by far the most challenging concept, both from compiler and application developers' standpoints. Hence we primarily focus on the OpenMP's offloading directives in this paper. Till date (submission of this journal) we have covered 60% of the offload directives. We do not count `simd` directives as they are impossible to verify without looking at the code that's generated. In this journal, we showcase results on IBM Power 9 architecture (Summit) along with results on X86 architecture. We also discuss overhead associated to different directives and different implementations prevalent on different platforms that we have access to. Studies on overhead findings for combined constructs were a particularly interesting aspect of the OpenMP implementations as this could potentially change the code generation in comparison to their nested versions. Also, we report findings on how the OpenMP compilers use CUDA drivers and CUDA runtime APIs through execution traces.

The following are the main contributions of this journal:

- Identify the extent of OpenMP 4.5 offload support in available OpenMP 4.5 implementations such as GCC, Clang, XL and Cray CCE while identifying and reporting inconsistencies or bugs in specific compiler implementations.

- Evaluate the available OpenMP 4.5 compiler implemen-

PREPRINT

tations on ORNL Summit besides 3 other systems.

- Define and use a testing methodology to evaluate overhead of directives across different OpenMP 4.5 implementations.
- Report studies of how the OpenMP compilers use CUDA driver and CUDA runtime APIs via execution traces.
- Evaluate changes in runtime overhead while using combined constructs vs. nested constructs, as well as the effect of changing the number of teams/threads on the overhead.

The remainder of the paper is organized as follows: Section 2 summarizes related efforts in this area of work. Section 3 discusses the concepts of offloading directives in OpenMP 4.5. In Section 4.1 we describe our experimental setup and in Section 4.2 we describe our feature tests used to assess the extent of support for offloading features in different compilers such as Clang, XL, GCC and Cray. In Section 5 we analyze the overheads of the standalone and combined constructs directives presenting our findings and analysis. In Section 6 we summarize our findings and discuss next steps.

2. Related work

Related efforts include work that discusses the status of implementations of OpenMP 3.1 features and OpenACC 2.5 features with different compilers [17, 18]. Such work has both highlighted ambiguities in the specifications and reported compiler bugs thus enabling application developers to be aware of the statuses of compilers. Similarly [19] validates OpenSHMEM library API. This work, in addition to feature tests, also provides micro-benchmarks that can be used to analyze and compare performances of library APIs. This is of special interest when targeting different OpenSHMEM library implementations on varying hardware configurations. Work in [14, 20] presents validations of implementations of OpenMP 2.0 features, which was further extended and improved in [13] to develop a more robust OpenMP validation suite and provided up-to-date test cases covering all the features until OpenMP 3.1. Since 2013, OpenMP can support heterogeneous platforms and the specification was extended with newer features to offload computation to target platforms.

The parallel testsuite [21] chooses a set of routines to test the strength of a computer system (compiler, run-time system, and hardware) in a variety of disciplines with one of the goals being to compare the ability of different Fortran compilers to automatically parallelize various loops. The Parallel Loops test suite is modeled after the Livermore Fortran kernels [22]. Overheads due to synchronization, loop scheduling and array operations are measured for the language constructs used in OpenMP in [23]. Significant differences between the implementations are observed, which suggested possible means of improving future performance. A microbenchmark suite was developed to measure the overhead of the task construct introduced in the OpenMP 3.0 standard, and associated task synchronization constructs [24].

Other related efforts to building a testsuite include Csmith [25], a comprehensive, well-cited work where the authors perform a randomized test-case generator exposing compiler bugs using differential testing. Such an approach is quite effective to detecting compiler bugs but does not quite serve our purpose since it is a challenge to automatically map a randomly generated failed test to a bug that actually caused it. We do not plan to apply Csmith concepts to our project just yet as we are dealing with compiler implementations that are not fully matured and it requires frequent communications with vendors in terms of reporting bugs and reusing next versions of compilers to identify if the issues are fixed. We also use combined and composite directives in our tests that need to be tested in order to mark them as compiler or runtime errors.

LLVM has a testing infrastructure [26] that contains regression tests and whole programs. The regression tests are expected to always pass and should be run before every commit. These are a large number of small tests that tests various features of LLVM. The whole program tests are referred to as the *LLVM testsuite*. The tests itself are driven by *lit* testing tool, which is part of LLVM. The LLVM testsuite itself does not contain any OpenMP accelerator tests excepting a very few tests on offloading and tasking. OpenMP 4.5 feature tests and reporting is covered in [27]. These test try to comprehensively cover the new directives in OpenMP 4.5 with all combination of clauses. In [16] we had evaluated the level of support for 4.5 features in multiple compilers and on a variety of systems. More specifically the features that we focused on included testing `target`, `target data`, `map`, `target enter data`, `target teams distribute parallel for` and `target update` and its associated clauses.

The above mentioned references are closely related to the scope of our current manuscript that is focusing on building testsuite and measuring overheads. Other types of work include performance analysis of OpenMP offloading model on GPUs [28, 29] that narrates the usage of features for real and proxy applications and their impact on the overall performance of the scientific applications on heterogeneous systems. The scope of such work is quite different to ours. While these papers discuss performance analysis of implementations, our project focuses on the validation and verification of features agnostic to specific implementation strategies along with determining overhead by runtime.

3. Offloading in OpenMP 4.5

A major change introduced in OpenMP 4.0 and improved in 4.5 is offloading computation to devices. This new feature enables the possibility of executing code in one or multiple co-processor devices (or accelerators) while at the same time running classical pre-OpenMP 4.0 parallel code on the multicore processor. Offloading opens a new world of heterogeneous computation for application developers, still, it brings signification changes to the OpenMP execution and memory model. Specifically, the addition of a new independent execution environment for each of the offloading devices (e.g. a different memory address space, memory hierarchy or core's micro-architecture). Accel-

PREPRINT

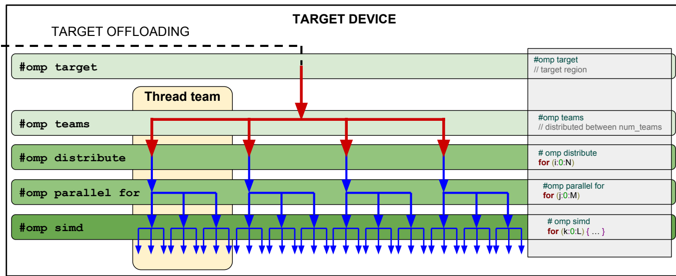


Figure 1: OpenMP 4.0+ Execution model.

erators such as GPGPUs and Xeon Phi coprocessors have heavily influenced the OpenMP definitions of execution models and memory models for offloading. OpenMP 5.0 is the most recently released specification that brings major improvements to the code offloading features. However, at the time of writing this paper (June 2019), there are no compilers that provide full OpenMP 5.0 support. In fact, implementations are still working towards having completing support for OpenMP 4.5, hence we focus on OpenMP 4.5 features that are of most importance to application developers and those that have been implemented by the majority of the compiler developers.

OpenMP uses a host-centric execution model. The *host* corresponds to the processor that initiates the program execution, and the *device* may correspond to the co-processor or an accelerator that is used for computing particular segments of code (kernels) in the program. The host will offload data and computation to one or more *target devices* using the `target` directive, wait for the execution to complete in these devices (possibly executing other code), and finally move results back to the host. This could happen many times throughout the program. Shared memory environments between accelerators and host is supported by the OpenMP specifications. In such case programmer might not need to explicitly move data back and forth between host and device.

Directives starting with `omp target` will indicate the region of code that will be considered for offloading to the accelerator device during runtime. Additionally, the programmer may define which data is to be copied back and forth, or allocated on the offloading device. The compiler converts these segments of code to kernel functions in the accelerator's instruction set, which can then be executed on the target device architecture. Offloading of code requires the target device to be present and supported by the compiler. If this is not the case the code should still run on the host. Hence, a host version of the target code will still be generated by the compiler. However, as far as the specification is concerned, there is no fall-back mechanism during runtime. This means that although the code could still be executed on a host with no target device on it, if the device is present but becomes unavailable during execution, or the execution on the device encounters some error, the program execution behavior will be unspecified.

The execution model of OpenMP 4.0+ continues to use the fork-join model, as well as the previously introduced tasking execution model. When the `target` construct is encountered, a new target task is created, enclosing all the target region. It is possible to specify dependencies between host tasks and target tasks. Moreover, the `nowait` clause can be used to asynchronously execute target tasks and host tasks or parallel regions. Within the target region the programmer can use the `teams`, `distribute`, `parallel for`, and `simd` to express parallelism in the fork-join model as depicted in Figure 1. The target region executes in a single thread. When the `teams` construct is encountered, a *league of thread teams* is created. The master threads of each team will execute the *teams region*. It is important to be aware that there is no implicit barrier at the end of a the teams region. The other three constructs `distribute, parallel for` and `simd` are loop constructs. The `distribute` construct will split the iteration space among all the league of thread teams. Hence, each master thread of each team will be `statically` assigned with a chunk of the iteration space for execution. The `parallel` construct allows parallelism within the *thread team*. The iteration space is split between all the threads within a team. Finally, the `simd` construct allows splitting an iteration space into SIMD lanes, as long as this is supported by the architecture.

On the host side, in order to support device offloading, a new device thread exists per physically available device. This thread is in charge of managing resources for that particular device, as well as handling communication between the host and this particular device. When a *target* region is encountered, the required data is mapped (read transferred) to the device and the created *target task* is scheduled into the selected *target device*. The caller thread could either block or continue the execution depending if the `nowait` clause is present. Once the accelerator has finished executing the code, the output data is usually mapped back to the device.

Regarding the device memory model, each target thread will have its own *target data* region that keeps track of memory mapping between host and device. Variables can be present in the host memory, the device memory or both. Synchronization of data, or data movement between the two environments can be (and should be) managed by the programmer. To do this, the `map` construct, together with a *map-type-modifier*, are used in the `target` directive, the `target data` directive, the `target enter/exit data` directives or the `target update` directive. The `map` construct specifies if data should be allocated (*alloc* modifier), deallocated (*delete* and *release* modifiers), or moved (*to*, *from* and *tofrom* modifiers) between host and device. If no modifiers are present, default mapping is in effect. Mapping of primitive types (e.g. `int` and `double`) uses the *to* modifier, while arrays and pointers use the *tofrom* modifier.

4. Testing Support for OpenMP 4.5 Offloading

4.1. Experimental Setup

For our experiments, we use three different system as our testbed environments: Summitdev [30], Summit [31], and an University of Delaware system called Eureka. Summitdev is an early access system that is one generation removed from Summit (worlds fastest supercomputer as of this writing), and

PREPRINT

features IBM S822LC nodes with two IBM POWER8 processors. Each processor has 10 cores, and each core has 8 hardware threads for a total of 160 threads per node. As target devices, there are 4 NVIDIA Tesla P100 GPUs per node. Summit has a hybrid architecture with each node containing multiple IBM POWER9 CPUs and NVIDIA Volta GPUs all connected together with NVIDIA's high-speed NVLink. Figure 2a shows a picture representation of a Summit node. Each node has over half a terabyte of coherent memory (high bandwidth memory + DDR4). This memory is addressable by all CPUs and GPUs. Additionally 800GB of non-volatile RAM is available (can be used as a burst buffer or as extended memory). The nodes are connected in a non-blocking fat-tree using a dual-rail Mellanox EDR InfiniBand interconnect.

Finally, a node of Eureka (at the University of Delaware) is depicted in Figure 2b. Each node contains 2 Intel E5-2670, with 8 dual SMT cores, for a total of 32 hardware threads. The system has a total of 64 GB, divided into 2 different banks of 32 GB directly accessible by each Intel chip. As acceleration devices, each node has 2 NVIDIA K40m cards with 12 GB of GDDR5 memory each.

For the runtime overhead analysis discussed in Section 5 we used a combination of the Eureka system and Summit, given that these two systems contain a variety of compilers that would allow the largest comparison set, while running on two different architectures, and generations of NVIDIA GPU Accelerators.

We currently run our tests in most of the compilers that already support for OpenMP 4.5 offloading constructs. This way we are able to analyze the validity of the tests and at the same time the behavior of each compiler's implementation for a particular construct under study. The compilers we use include GCC Version 7.1 through 8.2, IBM XL Version 16.1.1, and IBM XL Version 13.01, Clang CORAL Version 3.8, as well as Clang trunk versions 7.0.0 rc1, rc2, rc3 and release, and 7.0.1 rc1 and rc2.

4.2. Compilers' Level of Support

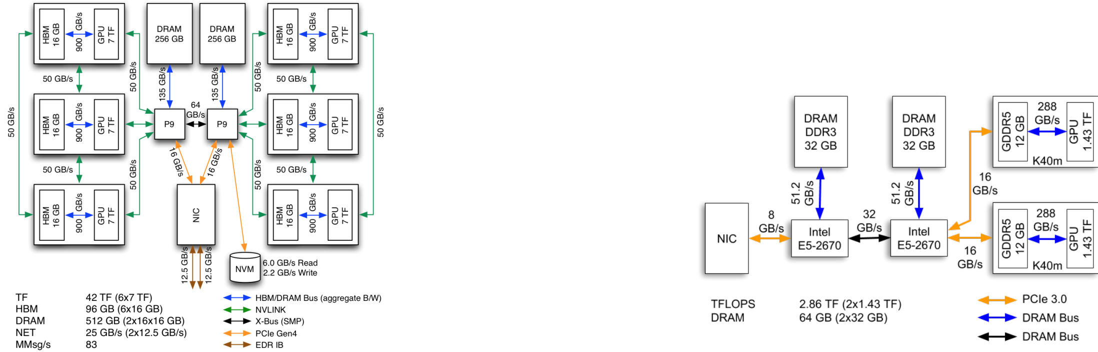
To ensure a broad coverage of the OpenMP offload directives, in this paper we present our analysis for `target`, `target data`, `target enter/exit data`, `target teams distribute`, `target update` and combined constructs. In order to assess the implementation state of different compilers that support OpenMP 4.5 offloading, it is necessary to study the level of support, in accordance to the specifications, of the different OpenMP constructs involving offloading features. Hence, we have created a set of tests that individually evaluate each of the constructs and their associated clauses, in accordance with the OpenMP 4.5 specification [2]. Each test was compiled and run with multiple compilers available to us on the test systems detailed in 4.1. These results are summarized in Table 1. Each column with a header containing a compiler's name and version represents a specific compiler running on a particular system. Due to the page limit consideration we picked a subset of the actual tests performed, running on the test systems. We have made available the complete list of results on our website [32]. The Table 1 omits tests that are currently in formulation or in the development phase and as a result have not passed the peer-review phase.

The tests undergo rigorous peer-review before we use them for our analysis, we have not elaborated on this review process as this is outside the scope of this paper. The version of the compiler is on the column header. Each row represents a test for a particular construct and clause. We differentiate between tests that pass with no issues (P), tests that pass compilation but have Runtime Errors (RE), and tests that have Compilation Errors (CE). In the case of compilation errors, some tests produce incorrect warning or error messages, while others will crash the compiler. Examples of compilation errors are tests 12 and 79, `is_device_ptr(var)` clause when compiled with GCC, which complains about `var` being mapped twice. Runtime errors could be due to OpenMP specification's compliance errors and program crashes. One example of a compliance error is test 13 `defaultmap` when compiled with XLC. `enum` variables, which are scalars, should be mapped as firstprivate by default when no explicit mapping is declared. However, they are mapped as `tofrom`. In general, the OpenMP 4.5 specification states are not mapped from the host to the device, but they have a data sharing attribute of `firstprivate` instead. Additionally, test 47 for the directive `target update` and the `devices` clause has a runtime error. The analysis of the exact cause of the failure is outside the scope of this paper.

here are test cases that do not yield to an error, but that result in a warning message due to unexpected behavior that is not necessarily against the specifications. An example occurs when using the `if(target:...)` clause and modifier with the `target teams distribute parallel for`. When the variable inside the clause evaluates to false, and given the `target` modifier, the parallel region should not change the number of threads. However, Some compilers would run the parallel region with a single thread as if the `if` clause would have affected both the parallel region and the target directive. While this cannot be considered an error this result is unexpected.

Clang running on Summit and summitdev, seems to be working well on the systems available to us. CORAL Clang version 3.8 which runs in summit and summitdev is different to the Trunk Clang version. Merging changes made to the CORAL clang is a work in progress. There are some errors that only affect Summit, which might indicate issues with the system-compiler combination or a version difference that was not reported due to the independent development process of the CORAL clang compiler. GCC offers an extensive support as well, but we see issues with `target data use device pointer`. However by providing timely feedback to the vendors we hope that a quick resolution is achieved. Most of the errors reported by these tests result in bug reports submitted to the related compiler vendor. Vendors have been aware of these problems and they have been actively working on solving them through the development of this work. Instances where the OpenMP Specification has been ambiguous - we have approached the OpenMP specification committee for more clarification. We anticipate newer versions of the specification to reflect comments/suggestions from our discussions with the com-

PREPRINT



(a) Summit: Power9 CPU and NVIDIA Volta V100s GPUs.

(b) Eureka: X86 Intel CPU with NVIDIA GPUs

Figure 2: Node Structures of Summit and Eureka

mittee members.

5. Overhead Comparison of OpenMP 4.5 Offloading

5.1. Methodology

Another aspect that is important to an OpenMP user or an application developer is to understand the overhead introduced by the translation between OpenMP clauses and the actual code that runs on the machine. Such overhead depends on many factors of the compilation process as well as the runtime necessary to support the OpenMP programming model. In order to support OpenMP, the compiler will look for annotations (Directives) in the code, and will replace them with runtime calls and additional code generation. If the annotation contains a region of code, the compiler will apply an outlining technique that consists of enclosing the region of code and placing them into a function that is later on hooked into the OpenMP runtime. In the case of offloading, the OpenMP compiler translates the region of code enclosed by the `target` directive to outlined device code and use multiple runtime calls to support code execution, data movement and any other required functionality. It is then important for application developers to understand the overhead introduced by the runtime calls as well as the outlining mechanisms. Such information is valuable for application developers as it gives an idea of the additional cost that is introduced by the compiler's translation of OpenMP directives. Overhead measurement, in conjunction with the support status of OpenMP 4.5 clauses described in Section 4, is also useful to understand the maturity of an implementation, as well as providing more insights to the developers about the compiler selection for their applications. Furthermore, such information could be used as a parameter for cost models of code offloading to accelerators. In addition to how the different clauses affect such overhead, it is also an interesting analysis to understand if the runtime overhead scales with respect to the requested hardware resources of the acceleration device, or it remains the same.

To this end, we intend to compare the runtime execution times of different OpenMP implementations, as well as the impact of the different OpenMP clauses. Additionally, we have studied how the OpenMP compiler uses the CUDA driver and CUDA runtime APIs through the execution traces. However, while CUDA has a well defined set of tools to profile it, currently, there is no standardized tools ecosystem that can provide precise information regarding OpenMP runtime execution times overall. Although this is likely to change with the introduction of the OpenMP Tools (OMPT) in OpenMP 5.0, this is currently not widely available across all the compilers we have evaluated, nor it is supported by the different profiling tools. Therefore for this work, we have proposed a methodology that isolates the OpenMP runtime as much as possible, by removing the device execution time, as well as the CUDA related execution time. Here we propose an indirect methodology to measure runtime overhead for offloading clauses in OpenMP.

Following are some assumptions and observations when measuring OpenMP directives overheads through our indirect method that should be considered when studying the result figures:

- Given the nature of offloading code to the device, there will be some overhead inherent in the underlying system (e.g. host-device interconnection, bandwidth). However, as long as we are running on the same system we expect this system overhead to be a constant across all implementations.

- Code generation for each compiler is different. The timing results we provide for Compiler A may not be the same for Compiler B as quality of device code generation as well as optimizations have an impact on the performance of a compiler, and this may vary from compiler to compiler. Therefore, the numbers we have provided are for comparison purposes only (same hardware different compilers) and to assess the quality of the runtime functionality for a given user code. The expectation is that the different results can provide an idea about the maturity of a particular runtime implementation, as well as the effect of using a clause on the overhead of such runtime.

- Due to the methodology used, it is necessary to amortize the effect of variables mapping and kernel computations. Most of the compilers do not allow empty target regions, hence it is necessary to provide a minimal code that does not represent a large segment of the measured time. We

PREPRINT

#	Test Name	(summit) CLANG 3.8.0 CORAL	(summit) XL 16.01	(summitdev) CLANG 3.8.0 CORAL	(summitdev) GCC 7.1.1	(summitdev) XL 13.01		Test Name	(Summitdev) GCC 7.1.1	(Summit) XLF 16.01
1	linked_list.c	P	P	P	P	P	51	offloading_success.F90	P	P
2	mmm_target.c	P	P	P	P	P	52	ompvv_template.F90	P	P
3	mmm_target_parallel_for_simd.c	P	P	P	P	P	53	target_data_if.F90	P	P
4	offloading_success.c	P	P	P	P	P	54	target_data_map.F90	P	P
5	offloading_success.cpp	P	P	P	P	P	55	target_data_map_components_default.F90	P	P
6	ompvv_template.c	P	P	P	P	P	56	target_data_map_components_from.F90	P	P
7	target_data_if.c	P	P	P	P	P	57	target_data_map_components_to.F90	P	P
8	target_data_map.c	P	P	P	P	P	58	target_data_map_components_tofrom.F90	P	P
9	target_data_map_array_sections.c	P	P	P	P	P	59	target_data_map_devices.F90	P	P
10	target_data_map_classes.cpp	P	P	P	P	P	60	target_data_map_from_array_sections.F90	P	P
11	target_data_map_devices.c	P	P	P	P	P	61	target_data_map_set_default_device.F90	P	P
12	target_data_use_device_ptr.c	P	P	P	CE	P	62	target_data_map_to_array_sections.F90	P	P
13	target_defaultmap.c	P	P	P	P	RE	63	target_depends.F90	RE	P
14	target_depends.c	P	P	P	P	P	64	target_device.F90	P	P
15	target_device.c	P	P	P	P	P	65	target_enter_data_allocate_array_alloc.F90	P	CE
17	target_enter_data_depend.c	P	P	P	P	P	66	target_enter_data_components_alloc.F90	P	P
18	target_enter_data_devices.c	P	P	P	P	P	67	target_enter_data_components_to.F90	P	P
19	target_enter_data_global_array.c	P	P	P	P	P	68	target_enter_data_devices.F90	P	P
20	target_enter_data_if.c	P	P	P	P	P	69	target_enter_data_if.F90	P	P
21	target_enter_data_malloced_array.c	P	P	P	P	P	70	target_enter_data_module_array.F90	P	P
22	target_enter_data_struct.c	P	P	P	P	P	71	target_enter_data_set_default_device.F90	P	P
24	target_enter_exit_data_depend.c	P	P	P	P	P	72	target_enter_exit_data_allocate_array_alloc_delete.F90	P	CE
25	target_enter_exit_data_devices.c	P	P	P	P	P	73	target_enter_exit_data_devices.F90	P	P
26	target_enter_exit_data_if.c	P	P	P	P	P	74	target_enter_exit_data_if.F90	P	P
27	target_enter_exit_data_map_global_array.c	P	P	P	P	P	75	target_enter_exit_data_module_array.F90	P	P
28	target_enter_exit_data_map_malloced_array.c	P	P	P	P	P	76	target_enter_exit_data_set_default_device.F90	P	P
29	target_enter_exit_data_struct.c	P	P	P	P	P	77	target_firstprivate.F90	P	P
30	target_firstprivate.c	RE	RE	P	P	P	78	target_if.F90	P	P
31	target_if.c	P	P	P	P	P	79	target_is_device_ptr.F90	P	CE
32	target_is_device_ptr.c	P	P	P	P	P	80	target_map_array_default.F90	P	P
33	target_map_array_default.c	P	P	P	P	P	81	target_map_components_default.F90	P	P
34	target_map_classes_default.cpp	P	P	RE	CE	RE	82	target_map_module_array.F90	P	P
35	target_map_global_arrays.c	P	P	P	P	P	83	target_map_pointer.F90	P	P
36	target_map_local_array.c	P	P	P	P	P	84	target_map_pointer_default.F90	P	P
37	target_map_pointer.c	P	P	P	P	P	85	target_map_program_arrays.F90	P	P
38	target_map_pointer_default.c	P	P	P	P	P	86	target_map_scalar_default.F90	P	P
39	target_map_scalar_default.c	P	P	P	P	P	87	target_map_subroutines_arrays.F90	P	P
40	target_map_struct_default.c	P	P	P	P	P	88	target_private.F90	RE	P
41	target_private.c	P	P	P	P	P				
42	target_teams_distribute.c	P	P	P	P	P				
43	target_teams_distribute_defaultmap.c	P	P	P	P	RE				
44	target_teams_distribute_device.c	P	P	P	P	P				
45	target_teams_distribute_is_device_ptr.c	P	P	P	P	P				
46	target_update_depend.c	P	P	P	P	P				
47	target_update_devices.c	P	RE	P	P	RE				
48	target_update_from.c	P	P	P	P	P				
49	target_update_if.c	P	P	P	P	P				
50	target_update_to.c	P	P	P	P	P				

Table 1: Level of support for multiple compilers and systems of OpenMP 4.5 offloading constructs. **Passed**(P) our tests, **Compilation Error**(CE), and **Runtime Error**(RE)

PREPRINT

```
OMPVV_INIT_TEST;
OMPVV_START_TIMER;
#pragma omp ...
{ OMPVV_TEST_LOAD; // if necessary }
OMPVV_STOP_TIMER;
OMPVV_REGISTER_TEST;
OMPVV_PRINT_RESULT;
```

Code 1: Overhead measurement testing methodology

have used a simple code that allows results to be driven mainly by the combination of the OpenMP runtime time, and the time taken by the device runtime, and not driven by the required code of the target region. CUPTI Allows us to distinguish between the two.

All of our tests have the structures presented in code listing 1. Slight modifications are applied depending on the nature of the construct being tested. For example the `firstprivate` or the `map` clauses require an extra variable. We use the CUPT library to obtain the execution trace of the CUDA-related functionality. We assume that the OpenMP runtime corresponds to anything else that is not CUDA.

The different parts of the tests preceded by `OMPVV_` are implemented in C macros to guarantee consistency. With `OMPVV_INIT_TEST` resetting all the timers for the current test. `OMPVV_START_TIMER` will make sure the CUPTI queue of events is empty, and enable a new set of traces and `OMPVV_STOP_TIMER` will wait for all the events queued in the CUPTI profiler and stop all the timers to obtain the final execution times. `OMPVV_REGISTER_TEST` will perform some preprocessing of the collected data to be able to parse it and process it later on. The tests are executed 100 times each to obtain significant statistical data. However we discard the max and min values of all runs to remove possible outliers in the data set. `OMPVV_TEST_LOAD` is applied to all the clauses that require a region of code. However, this code only instantiates a variable and increments it once, with the intention of making its contribution to the execution time as small as possible.

Finally OMPVV_PRINT_RESULT reports the total execution time in a predefined format. The C macros are translated to code that uses the CUPTI library available with the CUDA library.

A similar approach can be used to evaluate combined directives. According to the specifications, using combined constructs is semantically the same as nesting the different directives, as can be seen in code listing 2. However, as long as the semantic is kept, compilers can produce different codes. For example since there is no code between one directive and the next one, it is possible to simplify code generation by creating a single outlined region (instead of multiple nested outlined regions). We extend our methodology to also be able to compare if the overhead is different between using combined constructs

```
// combined constructs
# pragma omp target teams distribute
for (int i = 0; i < N; ++i) { ... }

// Nested constructs
# pragma omp target
  #pragma omp teams
      #pragma omp distribute
        for (int i = 0; i < N; ++i) { ... }
```

Code 2: Contrasting Combined and Nested Directives

and their nested counterpart across the different available compilers. Additionally, to restrict the possible number of teams and threads used by the two versions and across compilers, we assigned them by using the the *num_teams* and *num_threads* clauses. However, it is important to know that the actual values are still up to the compiler, as long as they are equal to or lower than the requested number of threads and teams. Combined directives offer a larger variety of possible combinations that can be used. We focused our attention on the effect of a single clause for combined clauses.

In addition, we explored the effect of requesting different number of teams and number of threads with respect to runtime overhead time when using the *target teams distribute* combined directive and the *target teams distribute parallel for* combined directive. This information would allow the user or an application to understand the effect that scaling a problem over a larger section of the acceleration device could have on the runtime overhead. As an example, these numbers or similar values (obtained through our testing methodology for a different system) could be used as part of a cost model that determines the appropriate distribution of work across the acceleration device. Additionally, these values play an important role in the analysis of sequential and overhead parameters of the Amdahl's law, when considering the benefits of parallelism for a particular application.

for the overhead calculations, we use Summit and Eureka as it allows us to compare a larger number of compilers, two different host architectures (POWER vs x86), and two different generations of NVIDIA accelerators (Tesla and Volta). We used *-O2* across all the experiments. However, OpenMP runtime are implemented by linking a pre-compiled library that contains the runtime code with the user generated binary. For this reason, the compilation flag should not have a considerable effect on the code. As explained before, we used the CUPTI library to obtain an execution trace of the CUDA library and CUDA driver, as well as memory copies and kernel execution times. The gray color on the results plot represent the CUDA execution time.

As compilers evolve towards a better implementation of the OpenMP standard, their implementations change from versions to versions. In order to do a fair comparison between compilers, we also have to consider not only the latest versions, but also their evolution across multiple versions. For GCC and Clang, as the two major open source compilers that implement OpenMP 4.5 offloading, there have been multiple versions and sub-versions releases that support OpenMP 4.5 offloading. This allows us to see their evolution over time. To this end, we used a large set of versions for these compilers on the Eureka cluster. For GCC versions 7.1 , 7.2 , 7.3 , 8.1 , 8.2. And for Clang versions 7.0.0 release candidates rc1 rc2 and rc3, and stable release; and 7.0.1 release candidates rc1 and rc2. We used all of these versions to run our overhead analysis and reported our findings. Despite these efforts, there was no statistically considerable changes in the execution times. For this reason we only provide results for a set of compilers. The extended version of this plots along with other plots are available on our website under publications tab. (Please look into the reports section).

5.2. Observation and Analysis

Our methodology has been designed to be able to offer a wide coverage of the offloading features, as well as the number of compilers. This section shows the results of our experiments to measure the overhead of the different directives and clauses for different compilers.

5.2.1. Offloading on Summit

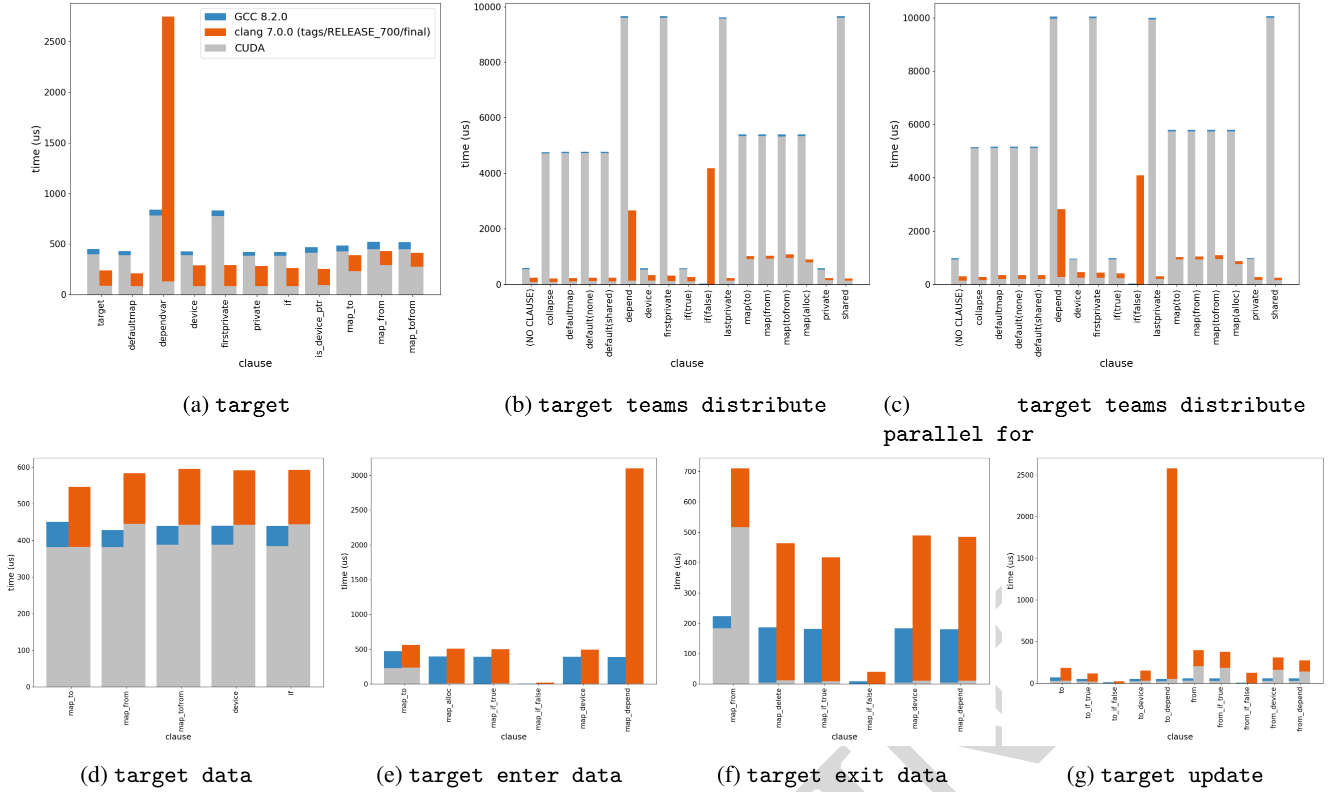
Figure 4 shows the overhead measurements using different offloading directives: `target`, `target data`, `target enter data`, `target exit data` and `target update` as well as the combined constructs `target teams distribute` and `target teams distribute parallel for`. The compilers available for this system are GCC 8.1.1, Clang (CORAL version, based off trunk version 3.8.0) and XLC 16.1.1.0. The x-axis represent the different clauses that are permissible with the directive and the y-axis gives the time in micro-seconds.

Across all the results we can see that XLC and Clang show considerably similar results. This is also the case for the CUDA traces studied, where the API calls and CUDA related calls happen at the same relative time and in the same order. There are only a few cases where this is not the case, some of which are mentioned later on.

On Summit, there is a clear difference between GCC, Clang and XL compilers with GCC performing significantly worse for the `target` directive, as can be seen in figure 4a. To further analyze where the differences are arising from, we profiled the calls using NVIDIA's CUPTI. CUPTI gives us a breakdown of the overhead times into CUDA calls and OpenMP runtime. We observe that with GCC, CUDA calls dominate the time taken for `target` directive. Further analysis showed us that specifically the CUDA memory allocation (`cuMemAlloc_v2`) and deallocation (`cuMemFree_v2`) functions were the most time consuming calls. In contrast, neither Clang nor XLC use CUDA memory allocation for the `target` directive. Since the tests validate the behaviour of the calls, meaning all implementations are correct, GCC seems to be doing a lot of extra memory allocations that may not be required (but not banned) by the OpenMP specification. In addition, it can be seen that the execution time of `cuLaunchKernel` is longer in both XLC and clang, in comparison to the GCC counterpart. While further studies of the source code is necessary to confirm this assessment, it is possible that certain parameters needed are sent as parameters of the kernel function. Such efforts have not been implemented in GCC, despite interest in the mailing list [33].

For the `target data` directive in figure 4d we see that all three implementations have comparable overhead for most

PREPRINT



(a) `target` (b) `target teams distribute` (c) `target teams distribute parallel for`

(d) `target data` (e) `target enter data` (f) `target exit data` (g) `target update`

Figure 3: Overhead measurement for offloading directives on Eureka cluster

clauses except the `map` clause with the `to` qualifier. More detailed profiling results show that both Clang and XLC spend more time in the OpenMP runtime. Unfortunately XLC being closed source with inadequate tools support (in our humble opinion) we cannot comment on which run-time calls are responsible for the uptick in overhead. Further studies of the clang and GCC compilers could provide more insights, but such efforts are out of the scope of this paper. Figure 4e shows a similar behaviour for `target enter data` with `map` clause with the `to` qualifier. This does not come as a surprise as both combinations involve setting-up the data environment before the computation is offloaded to the device, and as such must use similar/same code translation strategies.

Both `target enter data` with `map` clause with the `to` qualifier and `target exit data` (Fig 4f) with `map` clause with the `from` qualifier involve data transfer to the device and from the device respectively. Their detailed profiles show that along with the OpenMP runtime, the CUDA asynchronous memory copy from host to device (cuMemcpyHtoDAsync) (for `to`) and memory copy from device to host (cuMemcpyDtoH) (for `from`) contribute the most overhead. The `depend` clause for the `target enter/exit data` and `target update` seem to have no effect on GCC, but an effect on XLC and Clang compilers. This can be attributed to the differences in the implementations. The CUDA profiling details show that both Clang and XLC compilers rely on the cuEventQuery call to verify the completion of asynchronous memory copy calls while GCC uses synchronous memory copy functions.

The `target update` directive in figure 4g exhibits similar overhead profiles with more overhead associated with combinations that trigger data transfers.

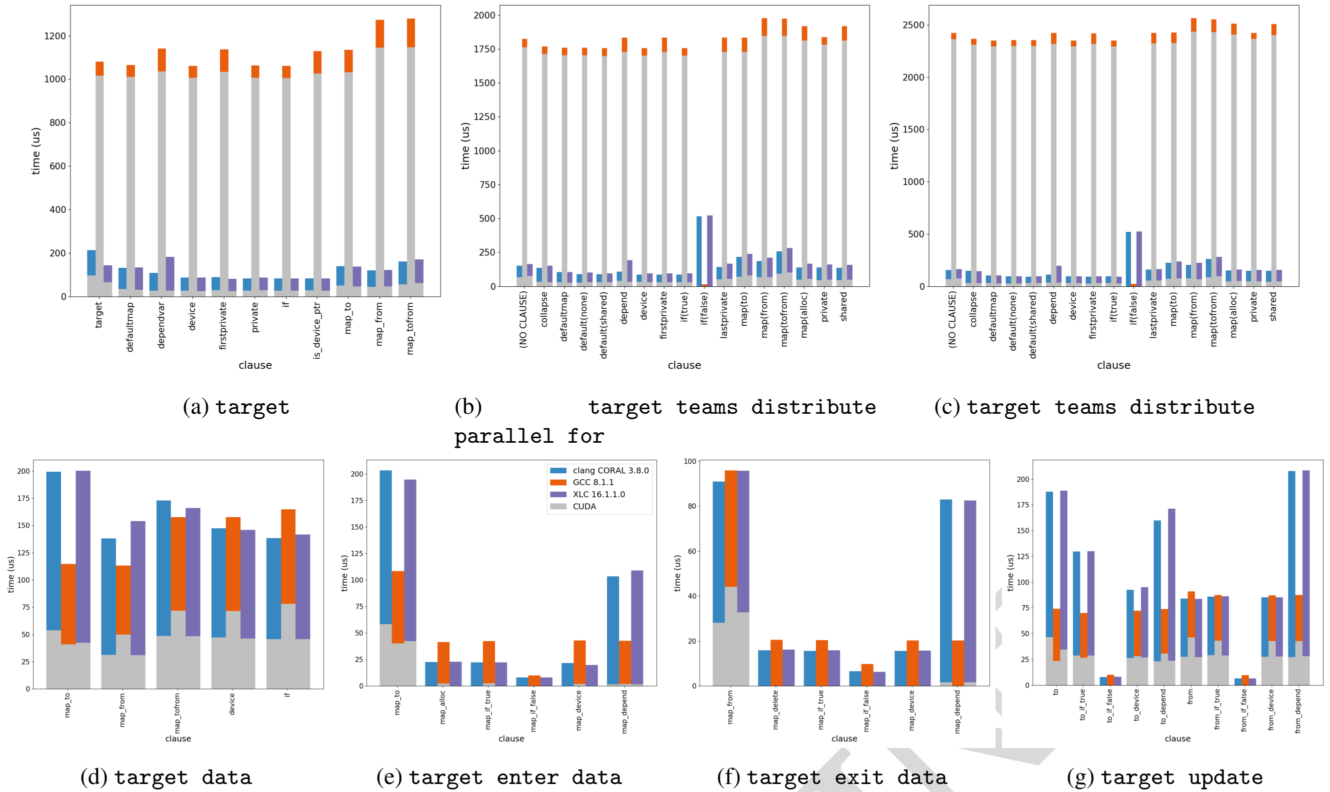
For `target teams distribute` directive in figure 4b and `target teams distribute parallel for` directive in figure 4c GCC spends a very high portion of the call doing CUDA calls for memory allocation and de-allocation, as it is the case of the `target` directive. In contrast both Clang and XLC use a combination of CUDA pointer manipulation functions and memory copy from host to destination to achieve the same results.

These two figures show an increasing cost with respect to spawning threads and teams, especially in GCC. However, the ratio between CUDA and OpenMP Runtime remains similar, and the behavior across the different clauses does not seem to change considerably.

5.2.2. *Offloading on Eureka*

We repeat the same experiments on the Eureka cluster built at the University of Delaware. Figure 3 shows the results for the Eureka system. Since access to Summit is restricted, most application developers rely on open-source implementations deployed on clusters (similar to Eureka) for development. Having data points to compare overheads of open-source implementations and their corresponding vendor implementations may prove helpful to application developers that would like to make performance-based design choices.

Here, we are comparing GCC 8.2.0 with Clang 7.0.0 (from



(a) `target` (b) `target teams distribute parallel for` (c) `target teams distribute`

(d) `target data` (e) `target enter data` (f) `target exit data` (g) `target update`

Figure 4: Overhead measurement for offloading directives on Summit

trunk). In the case of the `target` construct (figure 7b), Clang tends to incur less overhead than GCC for most cases except for the `depend` clause. The trend is seen across all the Clang `target` directives.

One of the main reasons behind it is that in Clang the `depend` clause binds the specified CUDA context to the calling CPU thread. This seems to trigger a lot of the OpenMP runtime activities.

The `target teams distribute` and `target teams distribute parallel for` both seem to spend a lot of time in CUDA memory allocation and freeing in GCC. Other constructs not discussed above are not significantly different in GCC and Clang.

Something that draws our attention is the large difference between GCC running in Summit with respect to overhead timing for the `target` directive, and the same version running in our Eureka Cluster (The reader is to be reminded that we ran all the different versions available to us at the time of running our tests, as in section 5.1). However, when teams and threads spawn, GCC on the Eureka cluster is actually worse for that case.

The execution traces reveal similar CUDA API calls, as well as execution order between these APIs. There seem to be little differences between the home-brewed clang version for summit, and the trunk version.

For GCC running in Eureka, there is a considerable difference in the overhead when using the `depend`, `firsprivate` and `lastprivate` clauses. We suspect that the required synchronization mechanisms for these clauses could cause this effect, however, further exploration is necessary to be able to draw larger conclusions.

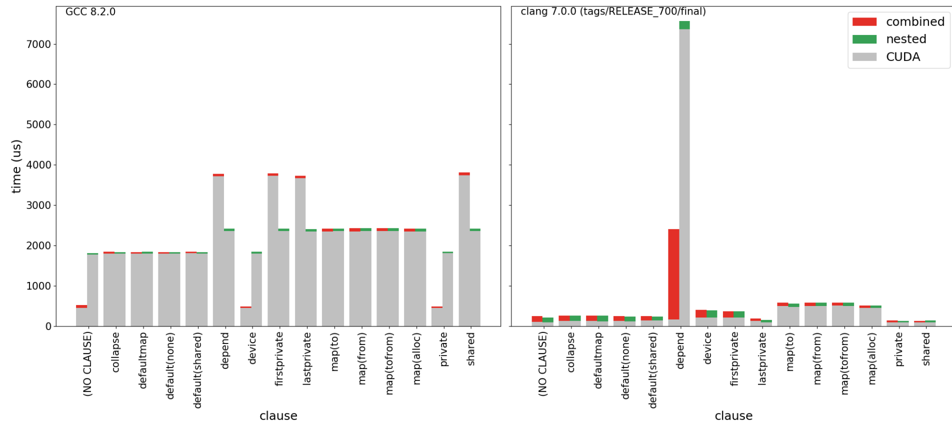
Versioning does not seem to have a large effect on the results either. As we did not observe drastic changes between versions of compilers, we have not reported results for every compiler version.

5.2.3. *Combined Constructs*

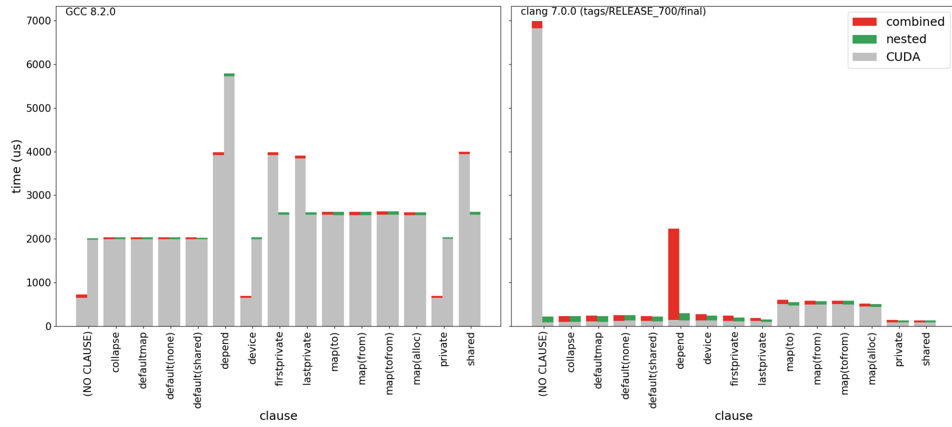
According to the OpenMP specifications, a combined constructs is:

> A construct that is a shortcut for specifying **one construct immediately nested inside another construct**. A combined construct is **semantically identical** to that of explicitly specifying the first construct containing one instance of the second construct and no other statements.

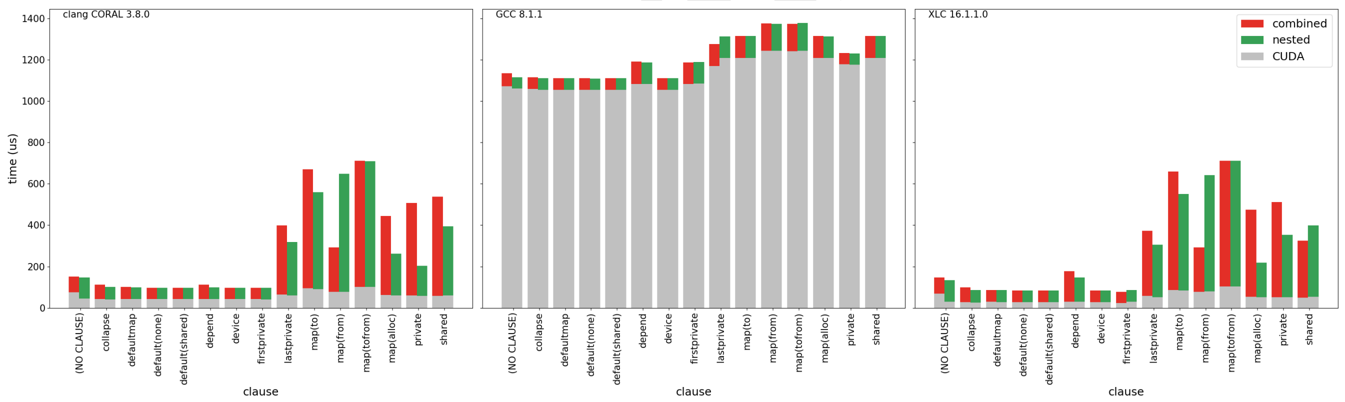
However, from code generation standpoint, and thanks to the lack of a block of code between the different nested constructs and the well structured generated code (i.e. no complex parallel vs sequential regions), it is possible to reduce the complexity of the resulting code. We developed a test following similar methods described above to study the runtime overhead timing of `target teams distribute` and `target teams distribute parallel for` constructs in combined and nested form. Results are reported in figure 5



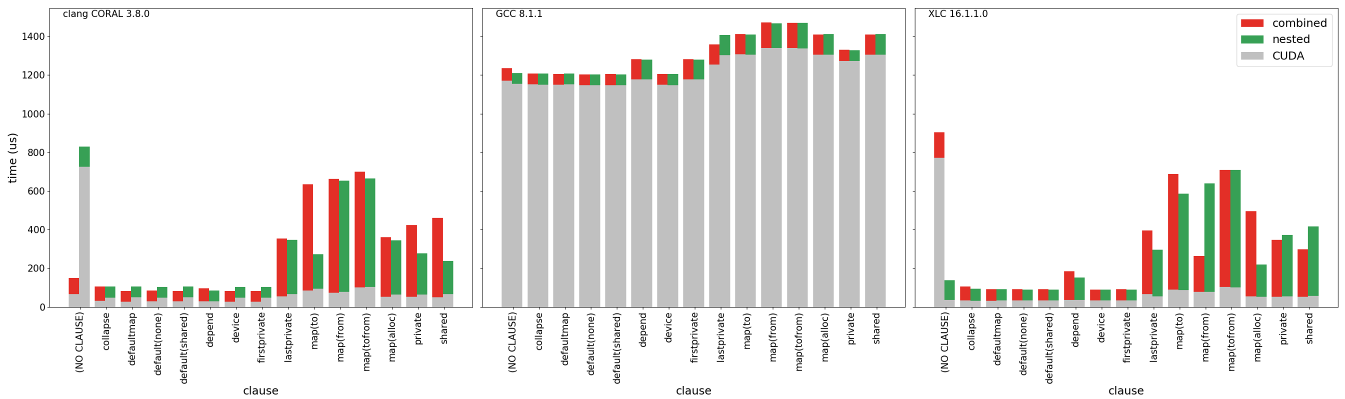
(a) `target teams distribute` Eureka



(b) `target teams distribute parallel for` Eureka

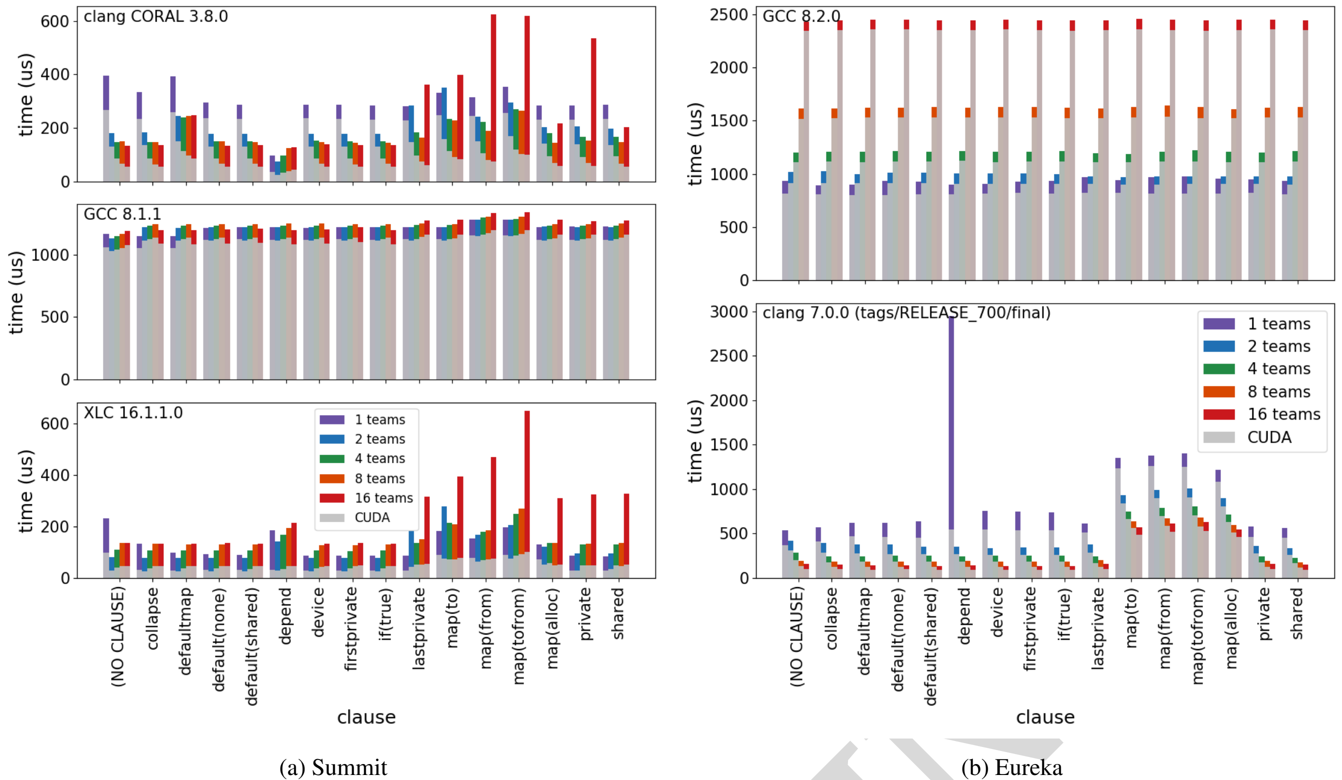


(c) `target teams distribute` summit



(d) `target teams distribute parallel for` summit

Figure 5: Comparing Combined directives vs. Nesting of those directives OpenMP directives



(a) Summit (b) Eureka

Figure 6: `target teams distribute` varying the number of teams. Effects of the number of teams on the overhead of the runtime on multiple compilers and systems

PREPRINT

Most of the results do not showcase major differences between combined and nested tests. Tests on the Eureka system show major differences in the `depend` clause across all the compilers for both `target teams distribute` and `target teams distribute parallel for`. In particular for the clang compiler, there is a large difference in the runtime portion of the overhead time (area not in gray). However, only for `target teams distribute` there is a considerable change in the CUDA behavior (area in gray). GCC on the other hand maintains the same runtime overhead across all the different clauses, but has larger variations in the CUDA for different clauses, without a clear preference between nested or combined.

On the contrary, Summit has a different behavior. The overall CUDA times have a low variation across almost all the tests between nested and combined. The only exception is for Clang and XLC when no clauses are used for the `target teams distribute parallel for`. On the other hand, the time taken by the runtime has a larger variability, but with no clear preference between nested or combined. Clauses that involve data mapping present a larger variability in the execution time.

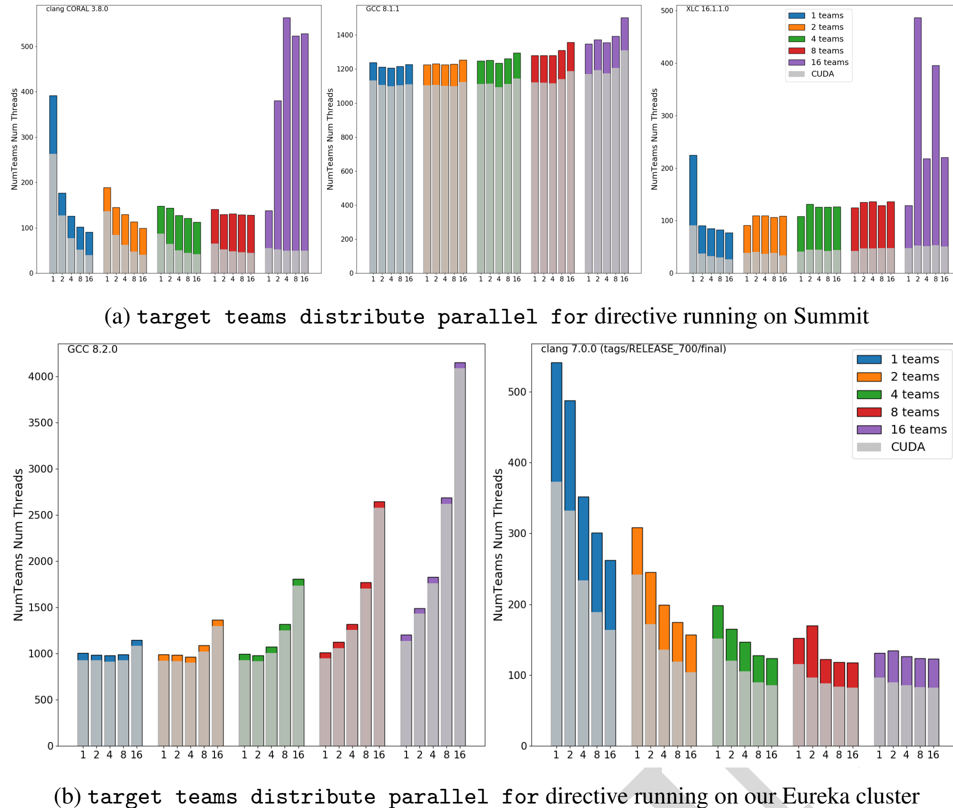
The data from our studies was not sufficient to determine the cause of the observed behavior between nested and combined as far as the OpenMP runtime is concerned.

5.2.4. The effect of number of teams and number of threads

Figure 6 shows how much the overhead changes according to the number of teams for the different compilers on both systems. For the case of GCC and Clang, on both systems, there are changes in the resulting overhead that depends on the number of teams. The variation of overhead seems to be higher for Clang, than for GCC. It is interesting to see that for team size of 1 Clang's overhead is more than any other larger team size. The exception is on Summit when teams are used along with the `map` clause. GCC has similar (but high) overhead on both Summit and Eureka. XLC's overhead trend on Summit gradually increases with an increase in the number of teams but we see a similar rise in the presence of `map` clause.

With respect to Figure 7, it provides an even more interesting perspective of the effect of the number of teams and the number of threads. Each bar of this plot corresponds to a single number of teams and a single number of threads. Figure 7 tries to capture the overhead changes when the number of threads vary with increasing number of teams. For Summit, the changes in overhead across all implementations are not as substantial as they are in Eureka. But the overhead trend for GCC is reversed on Eureka. On Summit, the overhead of GCC decreases with increase in number of teams and further increases with the size of the team (i.e. number of threads). On Eureka the overhead increases marginally with an increase in the number of teams however increases significantly with an increase in team size.

Clang on Eureka shows the most desirable overhead pattern with a decrease in overhead and increase in number of teams as well as the sizes of the teams. This suggests that during optimization of compute kernels compiled with GCC, it is a



(a) `target teams distribute parallel for` directive running on Summit

(b) `target teams distribute parallel for` directive running on our Eureka cluster

Figure 7: Effects of the number of teams and number of threads on the overhead of the runtime on multiple compilers and systems

distinct possibility that lowering the requested resources could lead to speedup.

6. Conclusion and Future Work

OpenMP's offloading capabilities will become more critical for an application to scale over large heterogeneous nodes that are commonly found and easily accessible lately. In this paper We provide a detailed analysis of how the different OpenMP compiler implementations support OpenMP 4.5 language features from three different perspectives: 1) testing of features OpenMP 4.5 offloading features described by the specification, 2) the runtime overhead and performance comparisons of all the OpenMP 4.5 offloading features, and 3) the runtime overhead and performance comparison of combined vs nested constructs as well as the effect of changing the number of teams and number of threads over this overhead. Additionally, we list ambiguities found while interpreting the description of the language features within the OpenMP specification as well as report possible errors found in the implementations to different vendors.

Although our current manuscript at the time of submission (April 2019) does not discuss 5.0 implementations as they are not available yet, we expect that the tests and the methodology to have a long term impact. Many of these tests will have a significant overlap between version 4.5 and 5.0, hence they will remain valid as is or with minor edits.

We also see these tests as a valuable resource for the OpenMP application developers. We are actively looking for common cases that we believe might be prone to implementation errors or that are important to applications. We have not discussed those kernels in this manuscript as they require further analysis before we can report our findings accurately. The tests and the overhead measurement methodologies could be used by HPC systems designers to assess the level of support of OpenMP offloading features on their platforms. In this work we also discuss overheads associated with different directives across their implementations prevalent on different platforms that we have access to. Studying how the OpenMP compilers use CUDA drivers and CUDA runtime APIs through execution traces was particularly useful to report findings. It is interesting to note that the wide differences in terms of overheads that are observed among the different compilers, as well as how some clauses could considerably hurt performance of applications. We believe these are some of the important takeaways of this work especially when there are not many such similar stories reported publicly for the users to read about. In addition, by comparing release versions of different compilers, there seem to be more focus on reaching compliance with respect to the specifications, than there is with respect to drastically changing the possible runtime implementations, leading to the same overhead across multiple compiler versions. We aim to make the our tests publicly available for anyone to use. Going forward, we also plan to interact with standard benchmarking bodies like SPEC/HPG that released SPEC ACCEL V1.0 [34, 35, 36] to donate our tests and kernels for potential inclusion in the next

release versions of SPEC OMP and SPEC ACCEL. Discussions are underway to eventually make the tests available as an official ARB testsuite. These tests will be used for acceptance testing in various facilities such as ORNL, LLNL, ANL to ensure the stability, performance, and functionality of future platforms at their respective locations.

7. Acknowledgements

This material is based upon work supported by the U.S. Department of Energy, Office of Science, 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 under contract number DE-AC05-00OR22725. We would also like to thank Tom Scogland from Lawrence Livermore National Laboratory for his contributions of OpenMP offloading usage in ECP applications and Hal Finkel from Argonne National Laboratory for his valuable input.

[1] Top500, Global supercomputing capacity creeps up as petascale systems blanket top 100, https://www.top500.org/news/global-supercomputing-capacity-creeps-up-as-petascale-systems-blanket-top-100/.

[2] OpenMP, Openmp 4.5 specification, http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[3] OpenACC, OpenACC, Directives for Accelerators, http://www.openacc.org/.

[4] NVIDIA, CUDA SDK Code Samples, http://developer.nvidia.com/cuda-cc-sdk-code-samples, accessed: 2017-02-03.

[5] OpenCL, OpenCL, https://www.khronos.org/.

[6] NVIDIA Thrust, https://developer.nvidia.com/thrust, accessed: 2017-02-03.

[7] H. C. Edwards, C. R. Trott, D. Sunderland, Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, Journal of Parallel and Distributed Computing 74 (12) (2014) 3202–3216.

[8] OpenMP, OPENMP 5.0 IS A MAJOR LEAP FORWARD, https://www.openmp.org/press-release/openmp-5-0-is-a-major-leap-forward/.

[9] M. P. Clay, D. Buaria, P. K. Yeung, Improving scalability and accelerating petascale turbulence simulations using openmp, http://openmpcon.org/conf2017/program/, to Appear (2017).

[10] M. Clay, D. Buaria, P. Yeung, T. Gotoh, Gpu acceleration of a petascale application for turbulent mixing at high schmidt number using openmp 4.5, Computer Physics Communications 228 (2018) 100–114.

[11] D. F. Richards, R. C. Bleile, P. S. Brantley, S. A. Dawson, M. S. McKinley, M. J. O?Brien, Quicksilver: A proxy app for the monte carlo transport code mercury, in: Cluster Computing (CLUSTER), 2017 IEEE International Conference on, IEEE, 2017, pp. 866–873.

[12] OpenMP, Openmp compilers, http://www.openmp.org/resources/openmp-compilers/.

[13] C. Wang, S. Chandrasekaran, B. Chapman, An openmp 3.1 validation testsuite, in: International Workshop on OpenMP, Springer, 2012, pp. 237–249.

[14] M. Müller, P. Neytchev, An openmp validation suite, in: Fifth European Workshop on OpenMP, Aachen University, Germany, 2003.

[15] NVIDIA, NVIDIA CUDA Profiling Tools Interface (CUPTI), https://developer.nvidia.com/CUPTI.

[16] J. M. Diaz, S. Pophale, K. Friedline, O. Hernandez, D. E. Bernholdt, S. Chandrasekaran, Evaluating support for openmp offload features, in: Proceedings of the 47th International Conference on Parallel Processing Companion, ACM, 2018, p. 31.

[17] C. Wang, R. Xu, S. Chandrasekaran, B. Chapman, O. Hernandez, A validation testsuite for OpenACC 1.0, in: Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International, IEEE, 2014, pp. 1407–1416.

[18] G. Kyle Friedline, Sunita Chandrasekaran, O. Hernandez, Openacc 2.5 validation testsuite targeting multiple architectures, In Proceedings of P3MA Workshop co-located with ISC 2017To appear.

[19] S. S. Pophale, A. Curtis, B. Chapman, S. Poole, Poster: Validation and verification suite for openshmem, in: Proceedings of the Seventh Conference on Partitioned Global Address Space Programming Model (PGAS 2013), 2013, pp. 257,258.

[20] M. S. Müller, C. Niethammer, B. Chapman, Y. Wen, Z. Liu, Validating openmp 2.5 for fortran and c/c++, in: Sixth European Workshop on OpenMP, KTH Royal Institute of Technology, Stockholm, Sweden, 2004.

[21] J. Dongarra, M. Furtney, S. Reinhardt, J. Russell, Parallel loops?a test suite for parallelizing compilers: Description and example results, Parallel Computing 17 (10-11) (1991) 1247–1255.

[22] F. H. McMahon, The livermore fortran kernels: A computer test of the numerical performance range, Tech. rep., Lawrence Livermore National Lab., CA (USA) (1986).

[23] F. J. Reid, J. M. Bull, Openmp microbenchmarks version 2.0, in: Proc. EWOMP, 2004, pp. 63–68.

[24] J. M. Bull, F. Reid, N. McDonnell, A microbenchmark suite for openmp tasks, in: International Workshop on OpenMP, Springer, 2012, pp. 271–274.

[25] X. Yang, Y. Chen, E. Eide, J. Regehr, Finding and understanding bugs in c compilers, in: ACM SIGPLAN Notices, Vol. 46, ACM, 2011, pp. 283–294.

[26] LLVM, Llvm Testing Infrastructure Guide, http://www.llvm.org/pre-releases/4.0.0/rc2/docs/TestingGuide.html#test-suite.

[27] J. M. Diaz, S. Pophale, K. Friedline, O. Hernandez, D. E. Bernholdt, S. Chandrasekaran, Evaluating support for openmp offload features, in: Proceedings of the 47th International Conference on Parallel Processing Companion, ICPP '18, ACM, 2018, pp. 31:1–31:10.
URL http://doi.acm.org/10.1145/3229710.3229717

[28] G.-T. Bercea, C. Bertolli, S. F. Antao, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, et al., Performance analysis of openmp on a gpu using a coral proxy application, in: Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems, ACM, 2015, p. 2.

[29] M. Martineau, S. McIntosh-Smith, W. Gaudin, Evaluating openmp 4.0's effectiveness as a heterogeneous parallel programming model, in: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2016, pp. 338–347.

[30] Oak Ridge National Lab, Ascending to summit: Announcing summitdev, https://www.olcf.ornl.gov/2017/02/28/ascending-to-summit-announcing-summitdev/.

[31] Oak Ridge National Lab, Summit, https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/.

[32] Jose Monsalve Diaz, Swaroop Pophale,Oscar Hernandez, David Bernholdt, and Sunita Chandrasekaran, Openmp 4.5 validation and verification suite, https://crpl.cis.udel.edu/ompvvsollve/.

[33] G. M. L. C. Philippidis), [patch,wip] use functional parameters for data mappings in openacc child functions, https://gcc.gnu.org/ml/gcc-patches/2017-12/msg01202.html.

[34] G. Juckeland, W. Brantley, S. Chandrasekaran, B. Chapman, S. Che, M. Colgrove, H. Feng, A. Grund, R. Henschel, W.-M. W. Hwu, et al., Spec accel: a standard application suite for measuring hardware accelerator performance, in: International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, Springer, 2014, pp. 46–67.

[35] G. Juckeland, A. Grund, W. E. Nagel, Performance portable applications for hardware accelerators: lessons learned from spec accel, in: Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International, IEEE, 2015, pp. 689–698.

[36] G. Juckeland, O. Hernandez, A. C. Jacob, D. Neilson, V. G. V. Larrea, S. Wienke, A. Bobyr, W. C. Brantley, S. Chandrasekaran, M. Colgrove, et al., From describing to prescribing parallelism: Translating the spec accel openacc suite to openmp target directives, in: International Conference on High Performance Computing, Springer, 2016, pp. 470–488.