# Evaluating Support for OpenMP Offload Features

Jose Monsalve Diaz[*]
University of Delaware
josem@udel.edu

Swaroop Pophale[*]
Oak Ridge National Laboratory
pophaless@ornl.gov

Kyle Friedline
University of Delaware
utimatu@udel.edu

Oscar Hernandez
Oak Ridge National Laboratory
oscar@ornl.gov

David E. Bernholdt
Oak Ridge National Laboratory
bernholdtde@ornl.gov

Sunita Chandrasekaran
University of Delaware
schandra@udel.edu

## ABSTRACT

The OpenMP language features have been evolving to meet the rapid development in hardware platforms. DOE applications tend to push the bleeding edge of features ratified in the OpenMP specification and tend to expose the rough edges of the features' implementations. The software harness on DOE supercomputers such as Titan and (upcoming) Summit include Cray, Clang, Flang, XL and GCC compilers. It is critical, especially for Summit, that the compilers support OpenMP offloading features. This paper focuses on evaluating support for OpenMP 4.5 target offload directives across compiler implementations on Titan and Summitdev, an early access system, which is one generation removed from Summit's architecture enabling application teams to test the systems' architecture. Our tests not only evaluate the OpenMP implementations but also expose ambiguities in the OpenMP 4.5 specification. We also evaluate compiler implementations using kernels extracted from production DOE applications. 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 are advertised as having only partial OpenMP 4.x support. We see this as a synergistic effort to help identify and correct features that are required by DOE applications and prevent deployment delays later on. Going forward, we also plan to interact with standard benchmarking bodies like SPEC/HPG to donate our tests and mini-apps/kernels for potential inclusion in the next release versions of SPEC OMP and SPEC ACCEL benchmark suites.

## CCS CONCEPTS

• **Computing methodologies → Parallel programming languages**; • **Hardware → Testing with distributed and parallel systems**; • **Software and its engineering → Dynamic compilers**; **Source code generation**;

## KEYWORDS

OpenMP 4.5, Offloading, Evaluation

[*]These authors contributed equally

https://doi.org/10.1145/3229710.3229717

**ACM Reference Format:**
Jose Monsalve Diaz, Swaroop Pophale, Kyle Friedline, Oscar Hernandez, David E. Bernholdt, and Sunita Chandrasekaran. 2018. Evaluating Support for OpenMP Offload Features. In *ICPP '18 Comp: 47th International Conference on Parallel Processing Companion, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3229710.3229717

## 1 INTRODUCTION

Top500 reports that eighty-six systems in the list are configured with accelerators and coprocessors, of which sixty use NVIDIA GPUs, twenty-one use Intel Xeon Phi cards, one uses AMD FirePro GPUs, one uses PEZY technology, and three systems use a combination of NVIDIA GPUs and Intel Xeon Phi coprocessors [28]. The trend towards heterogeneous architecture (CPUs+Accelerator devices) only seems to be strengthening with more systems using different types of cores with each year. The key advantage of heterogeneous systems is the performance for Watt contributed by accelerators in comparison to the traditional homogeneous systems that only use CPUs. These heterogeneous architectures offer tremendous potential in performance gains, but attaining that potential requires scientific applications of thousands or even millions of lines of code (LOC) 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 to abstract the hardware differences across different platforms and provide reproducible and identical behavior for applications. Directive-based programming models offer one of the best approaches to create reusable software without burdening the scientific application developers to learn about the programming paradigm or the intricacies of the hardware. Using directives, the programmers insert hints into a given region of code and the compiler automatically maps the code to efficient parallel code for the target system. Directives aim to achieve a performance portable code. Currently, the two popular directive-based programming models are OpenACC [21], OpenMP [23]. Besides directive-based approaches, other techniques to program accelerators include CUDA [17], OpenCL [22], NVIDA Thrust [1], and Kokkos [7].

OpenMP made a paradigm change to support heterogeneous systems and released a specification Versions 4.0 and 4.5 in 2013 and 2015 respectively. Features included directives such as `SIMD`, `target`, newer additions to `task` directive, such as `taskloop`, `taskloop simd`, `taskgroup` construct and newer clauses for tasks such as `priority`, 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 OpenMP 4.0 and 4.5 with language features to manage memory on systems with heterogeneous memories, with features for task reductions, extensions to the target construct along with several clarifications and fixes. OpenMP 4.5 and above allows programmers to use the same standard to program CPU, SIMD units, and the accelerators such as GPUs. This is currently being used by DOE and other scientific developers to port codes to heterogeneous systems. At Oak Ridge National Laboratory (ORNL) Pseudo-Spectral Direct Numerical Simulation-Combined Compact Difference (PSDNS-CCD3D) [4, 5], 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 [27].

Compiler support for OpenMP 4.5 has been increasing in the recent years [24]. GCC 7.1 (May 2017) provides support for OpenMP 4.5 in C and C++. IBM XL (Dec 2016) for 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 compiler supports OpenMP 4.5 for C. C++ and Fortran. For Cray systems the Cray Compiling Environment (CCE) 8.5 (June 2016) supports OpenMP 4.0 along with OpenMP 4.5 support for device constructs, Finally, LLVM Clang 3.9 released support for all non-offloading features of OpenMP 4.5, while a version of Clang that supports offloading features is currently under development.

As the device offload 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 be interpreted in multiple ways. Due to such ambiguities, vendors tend to interpret such descriptions differently and hence we find a particular feature implemented differently in different compilers, sometimes these differences are quite subtle but triggers a productive discussion to fix the description in the specification. Our previous publications have captured such discrepancies [15, 29]. OpenMP consumers benefit greatly from having a way to evaluate each compiler's implementation coverage of the given specification for each architecture where OpenMP-enabled applications are used.

The following are the main contributions of this work:

- Identify extent of OpenMP offload support (`target` directives) in OpenMP implementations such as GCC, Clang, XL and Cray.
- Analyze support for common code kernels identified across a range of DOE applications and test their support across all accessible OpenMP implementations.
- Identify and report inconsistencies or bugs in specific implementations to their respective compiler developers.
- Present performance data for different directives across different OpenMP implementations.

The remainder of the paper is organized as follows: Section 2 discusses the concepts of offloading directives in OpenMP 4.5. Our experimental setup is described in 3. We discuss our tests and findings on the extent of the support for offload features in different compilers such as Clang, XL, GCC and Cray in Section 4. We identify some common code snippets/kernels across applications in Section 5 and test their support across different compilers. In Section 6 we analyze the overheads of the `target` directives and present our findings and analysis. In Section 8 we summarize our findings and discuss next steps.

## 2 OFFLOADING IN OPENMP 4.5

One major change that was introduced in OpenMP 4.0 was offloading code to accelerators. 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. Code 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 microarchitecture). Accelerators such as GPGPUs and Xeon Phi coprocessors have heavily influenced the OpenMP definitions of execution models and memory models for offloading.

OpenMP uses a host-centric execution model. The *host* corresponds to the processor that initiates the program execution, and the *device* corresponds to the co-processor or acceleration device that is used to improve the computation of particular segments of code in the program. At some point during the execution, the host will offload data and computation to one or more *target devices*, 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.

As in a directive based programming models, the programmer provides hints to the compiler by using OpenMP `target` constructs. These directives will indicate the region of code that will be considered for offloading to the accelerator device during runtime. Additionally, the programmer uses directives to 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 fails, the program execution could still fail.

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. On the other hand, within the target region, the programmer can use the `teams`, `distribute`,

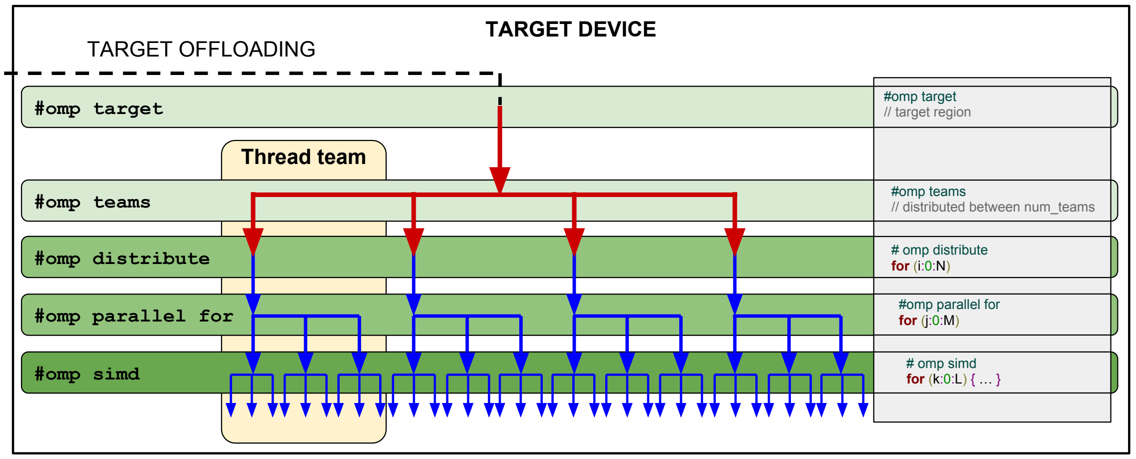


**Figure 1: OpenMP 4.0+ Execution model.**

`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, the device 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.

## 3 EXPERIMENTAL SETUP

For our experiments, we use Titan Cray XK7 [20], Summitdev [18] and Summit [19] as our testbed environments. Each Titan node uses a 16-core AMD Opteron x64 CPU and has access to 32 GB + 6 GB of DRAM. In addition, each node has one NVIDIA Kepler K20X GPU. Summitdev is an early access system that is one generation removed from OLCF's next big supercomputer and 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 2 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.

Because of page limit consideration we picked a subset of the actual tests performed. Our complete list is available on our website [8]. The table omits tests that are currently in the formulation or in the development phase and as a result have not passed peer-review. To ensure a broad coverage of the OpenMP offload directives, we present our analysis for `target`, `target data`, `target enter/exit data`, `target teams distribute` and `target update` in this paper. Each test is compiled with four different compilers that are available to us. These compilers already support for OpenMP 4.5 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

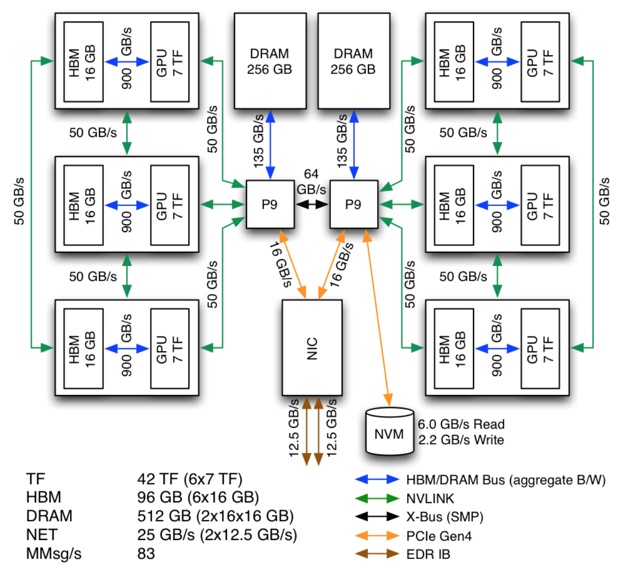


**Figure 2: A Summit Node.**

use include GCC Version 7.1.1, IBM XL Version 14.1 Beta 7, Cray CCE Version 8.6.5 and Clang Version 3.8.

Sections 4 and 5 looks at support for individual directives as well as common kernels across different OpenMP implementations. Section 4 gives information on overhead using different directives along with clause(s).

## 4 SUPPORT FOR OPENMP 4.5 OFFLOADING

In order to assess the implementation state of different compilers that support OpenMP 4.5, it is necessary to study the level of support 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 [23]. Each tests was compiled and ran with multiple compilers available to us in three different systems at ORNL as detailed in Section 3. These results are summarized in Table 1 where each column represents a specific compiler running on a particular system.

The tests undergo rigorous peer-reviews 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 Compiler Errors, some tests produce incorrect warning or error messages, while others will crash the compiler. Examples of compiler errors are compiling `is_device_ptr(var)` with GCC clause, which complains about `var` being mapped twice, and compiling `target enter data + if` with CCE crashes the compiler. Runtime Errors are differentiated between OpenMP specifications compliance errors and program crashes. One example of a compliance errors is `defaultmap` in XLC which maps `enum` variables as `tofrom` by default, even though scalar variables are not mapped by default and have a data sharing attribute of `firstprivate`. As of an example of a program crashing is `target data map (array sections)` compiled with CCE arrays. The `device` clause tests have been removed from the CCE compiler as their results are inconclusive given that Titan only has a single GPU per node available.

CCE seems to be the compiler with the most unsupported features. Important feature like array section mapping fails for multidimensional arrays, the `if` clause in some cases and `map to` of scalar values are also problematic. Clang, despite being a beta release, seems to be working well on the systems available to us. There are some errors that only affect Summit, which might indicate issues with the system-compiler interaction. GCC has extensive support as well, showing issues with `target data use device pointer`. Finally, XLC's `defaultmap(tofrom: scalar)` clause does not work for enum variables. Additionally, `target update devices` does not provide the right result either, and `target teams distribute parallel for if(parallel: ...)` complains about the number of thread not being as expected in the parallel region. The analysis of the exact cause of the failure is outside the scope of this paper. By providing timely feedback to the vendors we hope a quick resolution is achieved. Most of the errors shown by these tests resulted in bug reports one each compiler vendor or community.

## 5 SUPPORT FOR APPLICATIONS KERNELS

In this section we discuss few of the common kernels that we observed across a range of applications. Some of these contain multiple OpenMP directives and clauses. This is to verify that along with individual directives, the implementations can correctly support combination of OpenMP directives. We distill these application kernels into tests to ascertain that they produce the correct results.

### 5.1 Offloading to multiple devices

Code 1 tests the computational offloaded to different devices. This code distributes each row of a matrix to exactly one of the available devices. To perform this, we use a loop that iterates through the number of devices (lines 5-14), where each iteration performs data movement and computation in a specific device. First, the `target data` region maps a portion of the matrix to the device denoted by the value of `dev` (line 6). Then, the `target` region performs the computation on the device passed to the clause `device` (line 8).

Our first attempt defined the `target` region (line 8) without the map clause for mapping the `h_matrix` array (i.e. #pragma omp target device(dev)). Without the map clause map(from: h_matrix [dev*N:N]), the test reported success but at runtime the entire `h_matrix` was implicitly copied to each one of the devices. Since the intent was to divide data and minimize data movement we had to ensure that each device got a portion of the array (a row) rather than a copy of the `h_matrix`. To address this, we added the clause map(from: h_matrix[dev*N:N]) to line 8 such that the `h_matrix` array present implicitly in a `target` region is not larger than the original size already mapped in the outer region by the `target data` construct in line 7. It is important to clarify that the additional map(from: h_matrix[dev*N:N]) will not perform the data movement operation since the `h_matrix` array was already

| Test | Directive + Clause | CCE 8.6.5 Titan | clang 3.8.0 Summit | Clang 3.8.0 Summitdev | GCC 7.1.1 Summitdev | XLC 13.1.6.1 Summitdev | xlc 13.1.7.0 Summit |
|---|---|---|---|---|---|---|---|
| 1 | target + async | P | P | P | P | P | P |
| 2 | target data + if | P | P | P | P | P | P |
| 3 | target data map (array sections) | RE | P | P | P | P | P |
| 4 | target data map (classes) | RE | P | P | P | P | P |
| 5 | target data map + devices | - | P | P | P | P | P |
| 6 | target data map | P | P | P | P | P | P |
| 7 | target data + use device ptr | RE | P | P | CE | P | P |
| 8 | target (defaultmap) | RE | P | P | P | RE | RE |
| 9 | target + device | P | P | P | P | P | P |
| 10 | target enter data + async | P | P | P | P | P | P |
| 11 | target enter data + devices | - | P | P | P | P | P |
| 12 | target enter data (global array) | P | P | P | P | P | P |
| 13 | target enter data + if | CE | P | P | P | P | P |
| 14 | target enter data (malloced array) | P | P | P | P | P | P |
| 15 | target enter data (struct) | P | P | P | P | P | P |
| 16 | target enter exit data + async | P | P | P | P | P | P |
| 17 | target enter exit data + devices | - | P | P | P | P | P |
| 18 | target enter exit data + if | CE | P | P | P | P | P |
| 19 | target enter exit data + map (global array) | P | P | P | P | P | P |
| 20 | target enter exit data + map (malloced array) | P | P | P | P | P | P |
| 21 | target enter exit data (struct) | P | P | P | P | P | P |
| 22 | target + firstprivate | P | RE | P | P | P | RE |
| 23 | target + if | P | P | P | P | P | P |
| 24 | target + is device ptr | P | P | P | P | P | P |
| 25 | target + map (allocated array) | P | P | P | P | P | P |
| 26 | target + map (global arrays) | P | P | P | P | P | P |
| 27 | target + map (local array) | P | P | P | P | P | P |
| 28 | target + default map (pointer) | P | P | P | P | P | P |
| 29 | target + map (pointer) | P | P | P | P | P | P |
| 30 | target + default map (scalar) | P | P | P | P | P | P |
| 31 | target + default map (struct) | P | P | P | P | P | P |
| 32 | target + private | P | RE | P | P | P | RE |
| 33 | target teams distribute device | P | P | P | P | P | P |
| 34 | target teams distribute + is device ptr | P | P | P | P | P | P |
| 35 | target teams distribute parallel for (defaultmap) | RE | P | P | P | RE | RE |
| 36 | target teams distribute parallel for + devices | - | RE | P | P | P | RE |
| 37 | target teams distribute parallel for + firstprivate | P | P | P | P | P | P |
| 38 | target teams distribute parallel for + if (no modifier) | P | P | P | P | P | P |
| 39 | target teams distribute parallel for + if (parallel modifier) | RE | P | P | P | RE | RE |
| 40 | target teams distribute parallel for + if (target modifier) | P | P | P | P | P | P |
| 41 | target teams distribute parallel for + map (default) | P | P | P | P | P | P |
| 42 | target teams distribute parallel for + map (to) | RE | P | P | P | P | P |
| 43 | target teams distribute parallel for + private | P | P | P | P | P | P |
| 44 | target teams distribute | P | P | P | P | P | P |
| 45 | target update + async | P | P | P | P | P | P |
| 46 | target update + devices | - | P | P | P | RE | RE |
| 47 | target update + from | P | P | P | P | P | P |
| 48 | target update + if | P | P | P | P | P | P |
| 49 | target update + to | P | 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)**

```
int test_map_device() {
  int num_dev = omp_get_num_devices(), sum[num_dev], errors = 0;
  int* h_matrix = (int*) malloc(num_dev*N*sizeof(int));

  for (int dev = 0; dev < num_dev; ++dev) {
#pragma omp target data map(from: h_matrix[dev*N:N]) device(dev)
    {
#pragma omp target map(from: h_matrix[dev*N:N]) device(dev)
      {
        for (int i = 0; i < N; ++i)
          h_matrix[dev*N + i] = dev;
      } // end target
    } // end target data
  }

  // checking results
  errors = 0;
  for (int dev = 0; dev < num_dev; ++dev) {
    sum[dev] = h_matrix[dev*N + 0];
    for (int i = 1; i < N; ++i)
      sum[dev] += h_matrix[dev*N + i];
    errors |= (dev * N != sum[dev]);
  }

  return errors;
}
```

**Code 1: Offloading computation to multiple devices**

mapped in the outer `target data` region. All the test platforms (Titan (CCE 8.6.5), Summitdev (Clang 3.8, GCC 7.1.1, and XLC 13.1.6.1) and Summit (Clang 3.8 and XLC 13.1.7)) could successfully handle the decomposition and produce correct results.

## 5.2 Handling dependencies

The specification defines a task as any specific instance of executable code and its data environment, which was generated when a thread encountered a `task`, `taskloop`, `parallel`, `target`, or `teams` construct [23]. Tasking constructs are important since they allow a programmer to orchestrate the parallelism of an application by expressing dependencies between its tasks. Code 2 shows a task graph composed of host and target tasks that have dependencies between each other. The resulting task graph is shown in Figure 3. First, the tasks defined in lines 6 and 10 are two host tasks that initialize the input data and define their results as an output data dependency using the clause depend(out: variable ). Then, an operation of data movement from the host to the device is performed

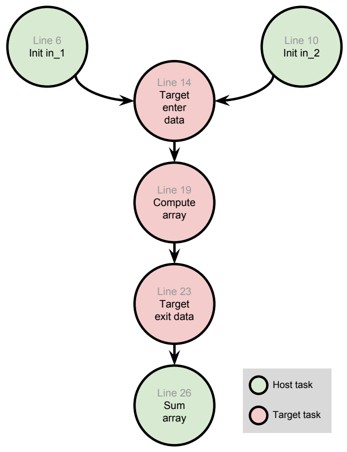


**Figure 3: Task graph created by Code 2**

using the asynchronous `target enter data` unstructured region defined in line 14. The asynchronous behavior is specified using `nowait` clause. In addition, line 17 specifies that the target task has input dependencies for the variables in_1 and in_2 (i.e. depend(in : in_1) and depend(in: in_2)). Finally, line 16 specifies that this target task generates array `h_array` as an output dependency (i.e. depend(out: h_array)).

Line 19 defines an asynchronous target region that performs the computation on the device. This target region defines an input/output dependency for the `h_array` variable using the depend( inout: h_array). In terms of the input dependency this means that the target region will wait until the task defined in line 14 is completed before starting its execution. It also means that the resulting `h_array` is an output dependency of this task. Similarly, the `nowait` clause specifies that the region will be executed asynchronously.

Line 23 specifies a data movement operation from the device to the host using an asynchronous `target exit data` construct. We specify an input dependency using the depend(inout: h_array). This will ensure that the data is only moved once the computation is done on the device. Also, as an output dependency, this means that the host task in line 26 will not start until the data movement is completed. Finally, we use the omp taskwait construct to signal the runtime that execution of the program must wait until all the tasks generated before this point are done.

Testing this on Summitdev led us to find that Clang produced incorrect results, although Cray, GCC, and IBM XL compilers passed the test. During the process of debugging, we determined that using the `nowait` clause was the culprit. After discussions with representatives of the OpenMP standard committee we were able to verify that our interpretation of the dependencies between tasks was correct. As a result, we filed a bug report with LLVM developers, providing a reproducible code of the error, the flags used for compilation, the details about our testbed, and a description of the faulty behavior. A possible workaround for this issue is to remove the `nowait` clause. This implies that the code will be executed in a serial fashion following the program execution order, which is not the intent of a user that wants to exploit task parallelism.

```
1  double sum = 0.0;
2  double* h_array = (double *) malloc(N * sizeof(double));
3  double* in_1 = (double *) malloc(N * sizeof(double));
4  double* in_2 = (double *) malloc(N * sizeof(double));
5
6  #pragma omp task depend(out: in_1) shared(in_1)
7  { for (int i = 0; i < N; ++i)
8      in_1[i] = 1; }
9
10 #pragma omp task depend(out: in_2) shared(in_2)
11 { for (int i = 0; i < N; ++i)
12     in_2[i] = 2; }
13
14 #pragma omp target enter data nowait \
15   map(alloc: h_array[0:N]) map(to: in_1[0:N]) \
16   map(to: in_2[0:N]) depend(out: h_array) \
17   depend(in: in_1) depend(in: in_2)
18
19 #pragma omp target nowait depend(inout: h_array)
20 { for (int i = 0; i < N; ++i)
21     h_array[i] = in_1[i]*in_2[i]; }
22
23 #pragma omp target exit data nowait \
24   map(from: h_array[0:N]) depend(inout: h_array)
25
26 #pragma omp task depend(in: h_array) \
27   shared(sum, h_array)
28 { for (int i = 0; i < N; ++i)
29     sum += h_array[i]; }
30
31 #pragma omp taskwait
32 errors = 2.0*N != sum;
```

**Code 2: Testing a task graph with dependencies.**

```
1  typedef struct node {
2      double data;
3      struct node *next;
4  } node_t;
5  void map_ll(node_t *head) {
6      if (!head) return;
7  #pragma omp target enter data map(to:head[:1])
8      while(head->next) {
9          // Note: using array dereference syntax, array section
             on leaf only
10         // Attachment is *not* explicitly guaranteed
11 #pragma omp target enter data map(to:head[0].next[:1])
12 #pragma omp target
13         {
14             head->next = cur;
15         }
16     }
17 }
18
19 void unmap_ll(node_t *head) {
20     if (!head) return;
21 #pragma omp target exit data map(from:head[0].data)
22     while(head->next) {
23         // Note: only copies back the data element to avoid
             overwriting next pointer
24 #pragma omp target exit data map(from:head[0].next[0].data)
25     }
26 }
```

**Code 3: Mapping linked list to device**

### 5.3 Mapping linked-list to device

The linked list is a data structure very commonly used in HPC applications. One potential way of mapping it to the device environment is shown in Code 3. The *map_ll* function (line 5) uses `target enter data` directive to first map the *head* of the linked list and subsequently the pointer to the next link using array dereferencing syntax. The *unmap_ll* (line 19) function explicitly copies the data using map-type *from* with `target exit data map`.

A correct mapping of a linked list on the device with explicit attachment is shown in Code 4. The code shows a correct way to map and modify a linked list on a device by using the `target enter data` directive. The only downside is that the explicit mapping is slower and not intuitive. Code 4 is part of a test that was developed from analyzing an ECP application that uses linked lists. The

```
 1 typedef struct node {
 2     double data;
 3     struct node *next;
 4 } node_t;
 5
 6 void map_ll(node_t *head) {
 7     if (!head) return;
 8 #pragma omp target enter data map(to:head[:1])
 9     while(head->next) {
10         // Note: explicit attachment
11         node_t *cur = head->next;
12 #pragma omp target enter data map(to:cur[:1])
13 #pragma omp target
14         {
15             head->next = cur;
16         }
17     }
18 }
```

**Code 4: Explicitly mapping linked lists to device**

function *map_ll* at line 6 accepts the *head* of the linked list as an argument. The pointer is mapped to the device environment (line 8) and then each node of the linked list is explicitly mapped using the `target enter data map` (line 12). The *unmap_ll* function remains unchanged. All the test platforms (Titan (CCE 8.6.5), Summitdev (Clang 3.8, GCC 7.1.1, and XLC 13.1.6.1) and Summit (Clang 3.8 and XLC 13.1.7)) could successfully handle creation, addition, deletion, modification and traversal of linked lists.

## 5.4 Deep Copy

Code 5 is another test case derived from a full-scale ECP application. It shows the use of `declare target` directive to ensure that procedures and global variables can be executed and data can be accessed on the device. When the C++ methods are encountered, device-specific versions of the routines are created that can be called from a target region. Deep copy is performed through the use of `target enter data` (lines 43 and 44) by first mapping the class and then the individual class members. This kernel failed to execute correctly on Titan (CCE 8.6.5) but was correctly interpreted and executed on Summitdev (Clang 3.8, GCC 7.1.1, and XLC 13.1.6.1) and Summit (Clang 3.8 and XLC 13.1.7).

## 6 PERFORMANCE COMPARISON

Another aspect that is important for an OpenMP user is to understand the overhead introduced by the translation between OpenMP clauses and the actual code that runs on the machine. A compiler that supports OpenMP directives implements an OpenMP runtime that acts behind the scenes to provide the desired functionalities. In the case of offloading, the OpenMP compiler translates the directives to device code and runtime calls. We look at the timing overhead as it provides valuable information that, in conjunction with the support status of OpenMP 4.5 clauses described in Section 4, helps the user decide on the appropriate compiler to use for their particular application. As performance is key, timing also gives us insights into the maturity of the implementations.

However, evaluating OpenMP runtime can be complicated especially when there is a lack of a standardized tools eco-system that can report at the granularity of the runtime calls. For our work we thus focus at the OpenMP directive level as we only intend to contrast the performance of different OpenMP implementations on the same platform. We propose a indirect methodology to measure runtime overhead for offloading clauses in OpenMP. This evaluation

```
 1 #define RealType double
 2 #define N 10
 3
 4 #pragma omp declare target
 5 class MyVector
 6 {
 7 public:
 8
 9     inline RealType operator()(int i, int j, int k) const
10     {
11         return X[k+Length[2]*(j+Length[1]*i)];
12     }
13
14     inline RealType& operator()(int i, int j, int k)
15     {
16         return X[k+Length[2]*(j+Length[1]*i)];
17     }
18
19     MyVector(int l, int m, int n)
20     {
21         Length[0] = l;
22         Length[1] = m;
23         Length[2] = n;
24         X = new RealType[l*m*n];
25     }
26
27     RealType *& getData() { return X; }
28
29     RealType * getData() const { return X; }
30
31     int getSize() const { return Length[0]*Length[1]*Length[2];
      }
32
33     int Length[3];
34     RealType * X;
35 };
36 #pragma omp end declare target
37
38 int main () {
39
40     MyVector gamma (N, N, N);
41     int size = gamma.getSize();
42
43 #pragma omp target enter data map(to:gamma)
44 #pragma omp target enter data map(to:gamma.X[0:size]) map(to:
      gamma.Length)
45
46 #pragma omp target
47     for(int i = 0 ; i < N ; i++)
48         for(int j = 0 ; j < N ; j++)
49             for(int k = 0 ; k < N ; k++)
50                 gamma(i,j,k) = 1.0;
51
52 #pragma omp target exit data map(from:gamma.X[0:size])
53
54     for(int i = 0 ; i < N ; i++)
55         for(int j = 0 ; j < N ; j++)
56             for(int k = 0 ; k < N ; k++)
57                 cout << gamma(i,j,k) << ". ";
58     return 0;
59 }
```

**Code 5: Deep Copy of C++ class members**

intends to provide an idea of the impact on the user code whenever target constructs and each of its clauses have been used. Following are some valid assumptions and observations when measuring overheads at OpenMP directive level:

- 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.
- It is not possible to make any decisions based on the actual timing numbers. These are for comparison purposes only (same hardware different compilers). The expectation is that the different results can provide an idea of the support and maturity of a particular implementation. Depending on which implementation has better support for the directives

```
OMPVV_INIT_TEST;
for (i = 0; i < NUM_REP; i ++) {
  OMPVV_START_TIMER;
  #pragma omp ...
    OMPVV_TEST_LOAD; // if necessary
  OMPVV_STOP_TIMER;
  OMPVV_REGISTER_TEST;
}
OMPVV_PRINT_RESULT;
```

**Code 6: Overhead measurement testing methodology**

used by their application, a user can choose one over the other.

- To amortize the effect of memory transfers and kernel operations we try to minimize their size and computational complexity. Thus, the results we obtain should reflect the code added by the compiler's OpenMP runtime.

All of our tests have the structures presented in listing 6. 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.

The different parts of the tests preceded by `OMPVV_` are implemented in C macros to guarantee consistency. With `OMPVV_INIT_TEST` the timer values, average, max and min are initialized. `OMPVV_START_TIMER` and `OMPVV_STOP_TIMER` do exactly that. `OMPVV_REGISTER_TEST` adds the current time to the average calculation as well as updates the max and the min. This process is executed `NUM_REP` (that for our case is 1002 times). 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. Finally OMPVV_PRINT_RESULT reports the average, maximum and minimum time for all the runs.

The C macros are translated to code that uses the `gettimeofday()` available in the *sys/time.h* library which has a resolution of $\mu s$ .

We used Summitdev for the overhead calculation as it allowed us to compare the larger number of compilers. We turn off all the compiler optimizations using `-O0` and `-g`. We discovered that each compiler has a different way of calling the CUDA compiler. Using the verbose mode we observed that GCC does not seem to call the compiler directly, and it is not possible to infer the flags. Clang uses `ptxas` with these flags: `-m64 -g --dont-merge-basicblocks --return-at-end -v` . Finally XLC calls `nvcc` with these flags: `-g -G -v`. For this reason, we took the results for the `target` directive shown in figure 4a, and we used `nvprof` to obtain the average execution time of each target region and removed it from the average execution time of our experiments. This way we obtained figure 4b.

Under these conditions, there is a clear disadvantage when using GCC over Clang and XL running on Power8 architecture. We believe that the former two compilers have better optimized implementations.It seems like XLC does not compile the kernel regions without optimizations, as their execution time is an order of magnitude lesser than xlc and GCC. We then see that clang and XLC are mostly similar across most of the constructs.

Other interesting results include overhead measurements when using the `depend` clause, execution times are much larger. It is likely that the required synchronization mechanisms for this clause comes with an additional cost. We also observe that using the `firstprivate` clause is expensive; their execution time is double compared to other clauses. The additional time is from mapping scalar values.

## 7 RELATED WORK

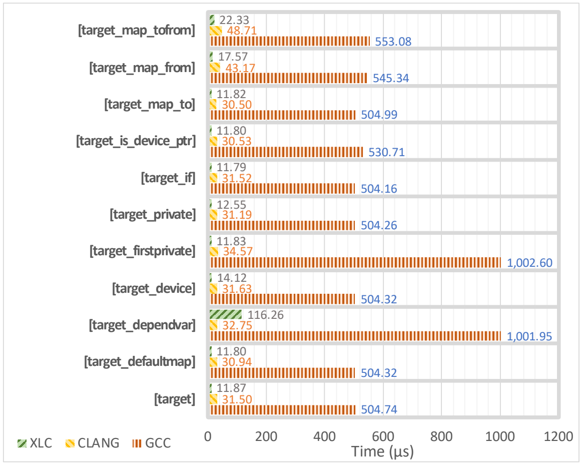
Related efforts include work that discusses the status of implementations of OpenMP 3.1 features and OpenACC 2.5 features with different compilers [12, 30]. Such work has both highlighted ambiguities in the specifications and reported compiler bugs thus enabling application developers to be aware of the statuses of compilers. Similarly [25] 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 [15, 16] presents validations of implementations of OpenMP 2.0 features, which was further extended and improved in [29] 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 [6] 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 [14]. Overheads due to synchronization, loop scheduling and array operations are measured for the language constructs used in OpenMP in [26]. 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 [3].

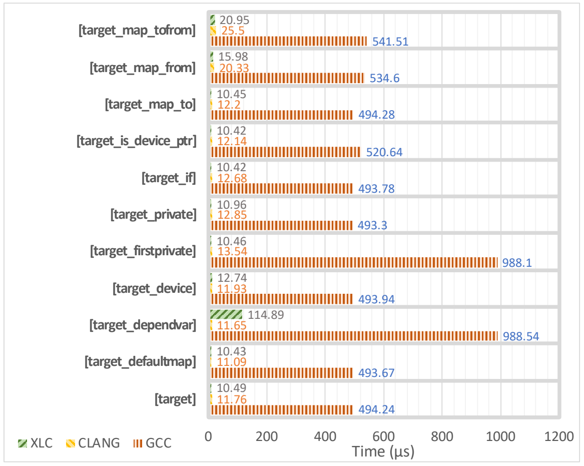
Other related efforts to building a testsuite include Csmith [31], a comprehensive, well-cited work where the authors perform a randomized test-case generator exposing compiler bugs using differential testing. Such an approach is quite effective to detecting compiler bugs but does not quite serve our purpose since it is hard to automatically map a randomly generated failed test to a bug that actually caused it. Thus we could say that our approach is complimentary to that of Csmith's approach. LLVM has a testing infrastructure [13] 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.

## 8 CONCLUSION AND FUTURE WORK

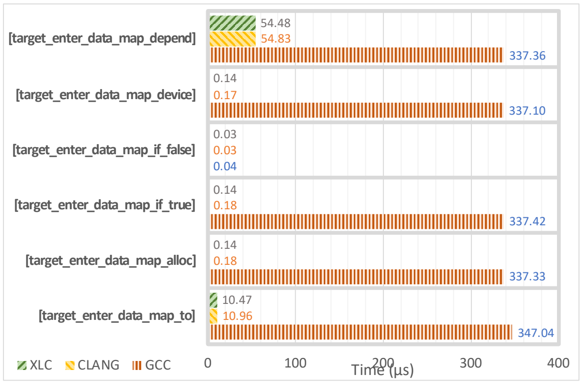
As we move to the CPU+device model, OpenMP's offloading capabilities will become more critical for an application to scale over fat nodes. Our detailed analysis on the support provided by different



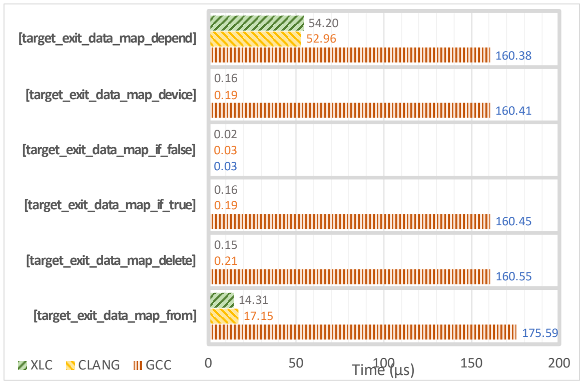
(a) **target directive**



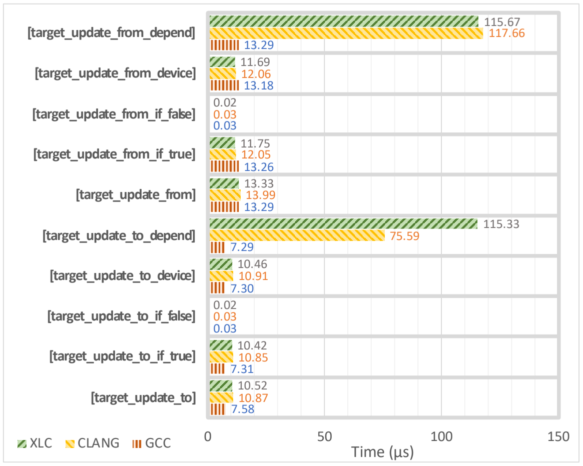
(b) **target directive no device region**



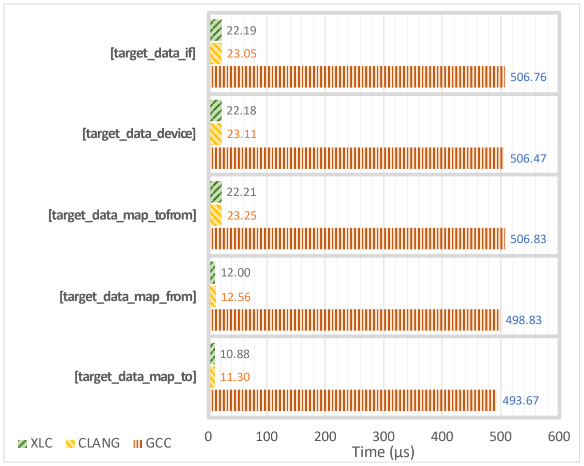
(c) **target enter data directive**



(d) **target exit data directive**



(e) **target update directive**



(f) **target data directive**

**Figure 4: Overhead measurement results**

OpenMP compiler implementations available for HPC applications gives insights into the maturity of these implementations as well as the ambiguities with respect to interpreting description of language features within the OpenMP specification. A by-product of this exercise is a collection of feature and performance tests segregated by the device construct that are a good foundation for a validation and verification testsuite. As next versions of the OpenMP specification

(OpenMP 5.0) will have a significant overlap, these tests any be valid as is or with some minor edits. We also see these tests as a resource for OpenMP users. Although there is a official resource for OpenMP 4.5 examples [2], the examples are not exhaustive in their coverage of directives in combination with all possible clauses defined in the specification. We also try to capture common cases that we believe might be prone to implementation errors or that are important to applications. Some of these cases are derived from large scale applications to test the common usage of OpenMP constructs, which may be using a combination of OpenMP directives and clauses. We also discuss overheads associated with different directives across their implementations prevalent on different platforms that we have access to. Although the majority of our current set of tests are implemented in C and C++, we plan to have Fortran versions in the near future.

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 [9–11] 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.

## 9 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.

## REFERENCES

[1] [n. d.]. NVIDIA Thrust. https://developer.nvidia.com/thrust. ([n. d.]). Accessed: 2017-02-03.

[2] OpenMP Architecture Review Board. [n. d.]. OpenMP Application Programming Interface. http://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf. ([n. d.]).

[3] J Mark Bull, Fiona Reid, and Nicola McDonnell. 2012. A microbenchmark suite for openmp tasks. In *International Workshop on OpenMP*. Springer, 271–274.

[4] MP Clay, D Buaria, PK Yeung, and T Gotoh. 2018. GPU acceleration of a petascale application for turbulent mixing at high Schmidt number using OpenMP 4.5. *Computer Physics Communications* 228 (2018), 100–114.

[5] M. P. Clay, D. Buaria, and P. K. Yeung. 2017. Improving Scalability and Accelerating Petascale Turbulence Simulations Using OpenMP. http://openmpcon.org/conf2017/program/. (2017). To Appear.

[6] Jack Dongarra, Mark Furtney, Steve Reinhardt, and Jerry Russell. 1991. Parallel Loops?A test suite for parallelizing compilers: Description and example results. *Parallel Comput.* 17, 10-11 (1991), 1247–1255.

[7] H Carter Edwards, Christian R Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216.

[8] Jose Monsalve Diaz, Swaroop Pophale,Oscar Hernandez, David Bernholdt, and Sunita Chandrasekaran. [n. d.]. OpenMP 4.5 Validation and Verification Suite. https://crpl.cis.udel.edu/ompvvsollve/. ([n. d.]).

[9] Guido Juckeland, William Brantley, Sunita Chandrasekaran, Barbara Chapman, Shuai Che, Mathew Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-Mei W Hwu, et al. 2014. 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, 46–67.

[10] Guido Juckeland, Alexander Grund, and Wolfgang E Nagel. 2015. Performance portable applications for hardware accelerators: lessons learned from SPEC ACCEL. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. IEEE, 689–698.

[11] Guido Juckeland, Oscar Hernandez, Arpith C Jacob, Daniel Neilson, Verónica G Vergara Larrea, Sandra Wienke, Alexander Bobyr, William C Brantley, Sunita Chandrasekaran, Mathew Colgrove, et al. 2016. From describing to prescribing parallelism: Translating the SPEC ACCEL OpenACC suite to OpenMP target directives. In *International Conference on High Performance Computing*. Springer, 470–488.

[12] GrahamLopez Kyle Friedline, Sunita Chandrasekaran and Oscar Hernandez. [n. d.]. OpenACC 2.5 Validation Testsuite targeting multiple architectures. *In Proceedings of P3MA Workshop co-located with ISC 2017* ([n. d.]). To appear.

[13] LLVM. [n. d.]. LLVM Testing Infrastructure Guide. http://www.llvm.org/pre-releases/4.0.0/rc2/docs/TestingGuide.html#test-suite. ([n. d.]).

[14] Frank H McMahon. 1986. *The Livermore Fortran Kernels: A computer test of the numerical performance range.* Technical Report. Lawrence Livermore National Lab., CA (USA).

[15] Matthias Müller and Pavel Neytchev. 2003. An openmp validation suite. In *Fifth European Workshop on OpenMP, Aachen University, Germany.*

[16] Matthias S Müller, Christoph Niethammer, Barbara Chapman, Yi Wen, and Zhenying Liu. 2004. Validating OpenMP 2.5 for fortran and c/c++. In *Sixth European Workshop on OpenMP, KTH Royal Institute of Technology, Stockholm, Sweden.*

[17] NVIDIA. [n. d.]. CUDA SDK Code Samples. http://developer.nvidia.com/cuda-cc-sdk-code-samples. ([n. d.]). Accessed: 2017-02-03.

[18] Oak Ridge National Lab. [n. d.]. Ascending to Summit: Announcing Summitdev. https://www.olcf.ornl.gov/2017/02/28/ascending-to-summit-announcing-summitdev/. ([n. d.]).

[19] Oak Ridge National Lab. [n. d.]. Summit. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/. ([n. d.]).

[20] Oak Ridge National Lab. [n. d.]. Titan supercomputer. https://www.olcf.ornl.gov/titan/. ([n. d.]).

[21] OpenACC. [n. d.]. OpenACC, Directives for Accelerators. http://www.openacc.org/. ([n. d.]).

[22] OpenCL. [n. d.]. OpenCL. https://www.khronos.org/. ([n. d.]).

[23] OpenMP. [n. d.]. OpenMP 4.5 Specification. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. ([n. d.]).

[24] OpenMP. [n. d.]. OpenMP Compilers. http://www.openmp.org/resources/openmp-compilers/. ([n. d.]).

[25] Swaroop Suhas Pophale, Anthony Curtis, Barbara Chapman, and Stephen Poole. 2013. Poster: Validation and Verification Suite for OpenSHMEM. In *Proceedings of the Seventh Conference on Partitioned Global Address Space Programming Model (PGAS 2013)*. 257,258.

[26] Fiona JL Reid and J Mark Bull. 2004. Openmp microbenchmarks version 2.0. In *Proc. EWOMP*. 63–68.

[27] David F Richards, Ryan C Bleile, Patrick S Brantley, Shawn A Dawson, Michael Scott McKinley, and Matthew J O?Brien. 2017. Quicksilver: A Proxy App for the Monte Carlo Transport Code Mercury. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE, 866–873.

[28] Top500. [n. d.]. 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/. ([n. d.]).

[29] Cheng Wang, Sunita Chandrasekaran, and Barbara Chapman. 2012. An openmp 3.1 validation testsuite. In *International Workshop on OpenMP*. Springer, 237–249.

[30] Cheng Wang, Rengan Xu, Sunita Chandrasekaran, Barbara Chapman, and Oscar Hernandez. 2014. A validation testsuite for OpenACC 1.0. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, 1407–1416.

[31] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 283–294.