OpenACC 2.5 Validation Testsuite Targeting Multiple Architectures

Kyle Friedline[1(✉)], Sunita Chandrasekaran[1], M. Graham Lopez[2], and Oscar Hernandez[2]

[1] University of Delaware, Newark, DE, USA
{utimatu,schandra}@udel.edu
[2] Oak Ridge National Lab, Oak Ridge, TN, USA
{lopezmg,oscar}@ornl.gov

Abstract. Heterogeneous computing has emerged as a promising fit for scientific domains such as molecular dynamics simulations, bioinformatics, weather prediction. Such a computing paradigm includes x86 processors coupled with GPUs, FPGAs, DSPs or a coprocessor paradigm that takes advantage of all the cores and caches on a single die such as the Knights Landing. OpenACC, a high-level directive-based parallel programming model has emerged as a programming paradigm that can tackle the intensity of heterogeneity in architectures. Data-driven large scientific codes are increasingly using OpenACC, which makes it essential to analyze the accuracy of OpenACC compilers while they port code to various types of platforms. In response, we have been creating a validation suite to validate and verify the implementations of OpenACC features in conformance with the specification. The validation suite also provides a tool to compiler developers as a standard for the compiler to be tested against and to users and compiler developers alike in clarifying the OpenACC specification. This testsuite has been integrated into the harness infrastructure of the TITAN and Summitdev systems at Oak Ridge National Lab and is being used for production.

Keywords: Programming models · Testsuite · Hardware · Validation

O. Hernandez—This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy 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). This paper is authored by an employee(s) of the United States Government and is in the public domain. Non-exclusive copying or redistribution is allowed, provided that the article citation is given and the authors and agency are clearly identified as its source.

1 Introduction

Hardware continues to evolve very rapidly expecting programmers to quickly deploy or redeploy scientific codes in order to exploit the rich feature set of these hardware platforms. These platforms are also becoming increasingly heterogeneous thus creating programming challenges and maintenance of single code bases. At the same time there is a constant demand for improved performance. Currently these heterogeneous platforms can be programmed using a variety of languages or models such as OpenACC [17], OpenMP [19], CUDA [15], OpenCL [18], NVIDA Thrust [3], Kokkos [8]. This paper focuses on OpenACC, which is an emerging directive-based programming model for traditional X86, accelerators such as GPUs, IBM Power processors, and, with OpenACC research compilers, FPGAs [4]. Since OpenACC can target more than just one or two platforms, it allows the programmers to maintain a single code base. Commercial compilers include support from PGI, Sunway Taihulight and Cray. Currently only PGI and GCC support OpenACC 2.5 (the version of the specification we are testing), though GCC has only partial support for the 2.5 feature set. Cray and Sunway provides support for OpenACC only until 2.0 features.

Open Source compilers include GNU Compiler Collection (GCC) with initial support for OpenACC 2.5. Academic compilers include Omni Compiler from Riken, OpenARC from ORNL, OpenUH from SBU and UH, RoseACC from LLNL, UDEL. More information on existing compilers can be found on the OpenACC webpage[1].

The model has been gaining wide adoption among the scientific community and is being used to accelerate scientific codes such as molecular dynamics, computational fluid dynamics, weather modeling to an accelerator. Instead of writing explicit code to offload or parallelize a given region of code, the programmer simply inserts compiler hints or directives into a C11, C++14 or a Fortran code, and the compiler offloads compute intensive portions of the code to an accelerator which could be multiple CPU cores and/or GPUs. As the OpenACC specification evolves and it's feature set is expanded, it is critical to ensure that the implementations of the features are conforming to the specifications and consistent with the definition of the features. It is quite common for different implementors to interpret the specifications differently. As a result there may be differing implementations of a particular feature defined in the specifications. Our previous publication on this effort [24] captured these discrepancies. The specifications has since evolved and there have been some major updates including providing support for both shared and discrete memory machines. So most, if not all, tests which this paper discusses are new and have been written to adhere to the current specifications.

This project creates a validation and verification testsuite where we construct a number of functional test cases to test several constructs such as the `parallel` or `kernel` constructs, clauses such as `async` or `reduction`, or combinations of occurrences of clauses on constructs. The testsuite is built to check

[1] https://www.openacc.org/tools.

for correctness and conformance of features to the OpenACC specifications. We will also demonstrate the results of the testsuite on multiple compilers/versions to compare implementations' adherence to the OpenACC specifications. This testsuite will enable the compiler implementors to improve the quality of their tools and ensure compliance of their implementations with the specifications of the language.

OpenACC has been adopted for large-scale scientific applications on accelerator-based supercomputers such as Titan at the Oak Ridge Leadership Facility (OLCF). These applications include: ACME Land Model and CAM/SE [21,22], One-Way Based Methods OWBM (Oil and Gas), Dalton [6], Maestro [14] (Astro-physics), GTC-p [5,9], and many are part of funded user programs such as INCITE [2] and CAAR [1]. These show that OpenACC has already afforded applications early successes at scale on heterogeneous machines such as Titan. Such large scale applications demand updates to OpenACC data clauses to allow deep copy. This feature is needed to handle complex data structures and how they are mapped to the limited memory capacity of discrete accelerators with disjoint memory address spaces. The OpenACC organization put together a technical report (TR-16-1) discussing challenges to support deep copy and also propose probable solutions [16]. Secondly, the increasing complexity of upcoming architectures means that OpenACC needs to be improved to handle and optimize for deeper node-level memory hierarchies; currently, the `acc cache` directive provides insufficient optimization and limits performance where memory capacity and bandwidth trade-offs are critical. Finally, the ability to better handle multiple accelerators per node will be critical to exploiting upcoming machines.

A comprehensive and community-driven OpenACC validation suite is an essential tool for computing facilities that must procure and evaluate both large production and experimental systems. As multiple compiler implementations are adapted to new architectures and updated to support evolving specifications, OpenACC consumers benefit greatly from having a way to evaluate each implementation's coverage of a given specification on each architecture where OpenACC-enabled applications are used. By having a community-driven validation suite that can be used in common by any vendor as well as the end users, application developers have a way to evaluate and push for consistency in functionality across implementations.

Since the tests are in a more matured state than it was earlier, we plan to release the testsuite by July 2017, just in time for the camera ready version of this paper. This testsuite will be released under a dual license scheme and we welcome contributions. One license will preserve the license used by the contributor, the second OpenACC license will ensure consistency in code version, and the running and reporting of results. Currently all OpenACC members can use and contribute to the testsuite.

The paper makes the following contributions:

– Develop test cases that can test on both shared and discrete memory models
– Identify and report compiler bugs and runtime errors

- Evaluate different compilers' implementations for its conformance to the OpenACC specifications
- Delivered initial release of OpenACC 2.5 Testsuite[2]

2 Overview of the Programming Model

The underlying goal of OpenACC is to deliver an API for parallelizing code targeting a generic heterogeneous architecture. With three layers of parallelism as well as a compute construct designated for compiler targeting of generic architectures, the model aims to abstract the architecture with minimal adjustments to the logic of the code thus allowing maintenance of a single code base. With often only minor adjustments to memory management near parallelized compute regions, the model accommodates both shared and discrete memory or any combination of the two across any number of devices.

The basic functionality of the model deals with specifying the compute intensive regions of code that needs to run on an accelerator as well as manage data on multiple devices. A great deal of this is managed by creating scopes with certain descriptors. `Parallel` or `kernels` directives need to be added to the code region that needs to be offloaded to the accelerator.

Since the last time this topic has been discussed (see [24]), the OpenACC specifications have undergone two major changes. OpenACC 2.0 version was released with features for queuing and resynchronization of asynchronous compute regions via the `wait` clause (see Code 1); dynamic data lifetimes via the `enter data` and `exit data` directives (see Code 2); asynchronous synchronizing of queues via the `async` clause on a `wait` directive (see Code 3); and function calls from compute regions via the `routine` directive (see Code 4).

Code 1. Resynchronization of Queues

```
#pragma acc parallel loop present(a[0:n], \
   b[0:n], c[0:n]) async(1)
for (int x = 0; x < n; ++x){
   c[x] = a[x] + b[x];
}
#pragma acc parallel loop present(d[0:n], \
   e[0:n], f[0:n]) async(2)
for (int x = 0; x < n; ++x){
   f[x] = d[x] + e[x];
}
#pragma acc parallel loop present(c[0:n], \
   f[0:n], g[0:n]) async(3) wait(1, 2)
for (int x = 0; x < n; ++x){
   g[x] = c[x] + f[x];
}
```

[2] https://github.com/OpenACCUserGroup/OpenACCV-V.

Code 2. Executable Data Directives

```
#pragma acc enter data copyin(data[0:n])
#pragma acc parallel loop present(a[0:n]) reduction(+:total)
    for (int x = 0; x < n; ++x){
  a[x] = a[x] * 2;
  total += a[x];
}
if (total < n){
  \\Conditionally updates data
  #pragma acc exit data copyout(data[0:n])
}
else {
  #pragma acc exit data delete(data[0:n])
}
```

Code 3. Asynchronous Synchronization of Queues

```
#pragma acc parallel loop present(a[0:n], \
  b[0:n], c[0:n]) async(1)
for (int x = 0; x < n; ++x){
  c[x] = a[x] + b[x];
}
#pragma acc parallel loop present(d[0:n], \
  e[0:n], f[0:n]) async(2)
for (int x = 0; x < n; ++x){
  f[x] = d[x] + e[x];
}
#pragma acc wait(1, 2) async(3)
#pragma acc parallel loop present(c[0:n], \
  f[0:n], g[0:n]) async(3)
for (int x = 0; x < n; ++x){
  g[x] = c[x] + f[x];
}
```

OpenACC 2.5, released in 2015, had a major update on the data management. In previous versions, data was managed by `copy`, `copyin`, `copyout`, and `create` data clauses as well as their respective counterparts, `present_or_copy`, `present_or_copyin`, `present_or_create`, `present_or_copyout`, that would check their presence on the device and allocate/copy as necessary. However, with version 2.5 data control was simplified by merging the data clauses and only supporting the functionality that tests for data's presence on device. Along with this, reference counting and conditional data updates with the `if_present` clause were added in version 2.5 to further simplify data management.

Code 4. Routine

```
int pow(int base, int exponent){
  returned = 1;
  for (int x = 0; x < exponent - 1; ++x){
    returned = returned * base;
  }
  return returned;
}
#pragma acc routine(pow) seq
#pragma acc parallel loop present(a[0:n])
for (int x = 0; x < n; ++x){
  a[x] = pow(a[x], 2);
}
```

To briefly discuss reference counting, this is managed by the compiler runtime libraries. Every time a variable is either used in an `enter data` directive or at the entrance to a data region, the variable's reference count is incremented. At either an `exit data` or at the exit of a data region, it is decremented. If before incrementation, the count is zero, the data clause is executed, either copying the data in or allocating the data on device. And if after decrementing, the count is zero, the data clause is again executed, either copying out the data or deallocating it on device.

One of the issues with this functionality is that it still leaves data validity very hard to determine due to data clauses potentially only altering reference counts instead of updating data. Also with the 2.5 version, the clause `if_present` for the `update` directive allows the update to be dependent on the presence of the data on device. This, in conjunction with the updated data management directives/clauses the user is able to exploit much finer control over the data environment.

3 Methodology

As mentioned in Sect. 1, we develop tests that target platforms with discrete or unified memory, or a combination of the two. This means that when a test is executed, the parallel regions of the code could be operating on a device with direct access to the host-processes memory. We also develop tests that address the dynamic execution order. The nature of parallel programming creates race conditions and difficulty with ordering and flow control. These, among other complications, pose difficulties to creating platform-agnostic tests. Furthermore, the desire of these tests is to find flaws in the compilation of the program and to do so we try to design tests in such a way that they can strain the limits of what the specifications dictate.

The constraints with the model often limit our ability to make the test as discerning as we might want. Although there are hundreds of test case scenarios, in this section we will illustrate these constraints using the example of testing the `create` clause.

According to the language specifications, the `create` clause only allocates the memory on the device; it does not copy the host's values for the data to the device. When testing this aspect of the language, there are some vague aspects of the execution of the clause. First, the code could be run on either a shared memory device, or a discrete memory device. This means that at the `create` call, if the application is running on a shared memory device, the data will already be present and the data clause will be ignored. On a discrete memory device, at the call to the `create` clause, the memory will be allocated on the separate memory device. The data is not copied and will be uninitialized in the device memory.

With these potentialities for possible execution methods, how does one test that the creation happened without the copy? To demonstrate the complications in writing tests that satisfy these requirements, let's look at an example for testing this clause on an `enter data` directive. The specifications state that when an array is in a create clause, that the data is allocated, but not copied. If we wanted to test this functionality, we might create something such as what is in Code 5.

Code 5. Enter Data Create V.1

```
double * a = (double *)malloc(n * sizeof(double));
double * a_copy = (double *)malloc(n * sizeof(double));
for (int x = 0; x < n; ++x){
  a[x] = rand();
  a_copy[x] = a[x];
}
#pragma acc enter data create(a[0:n])
#pragma acc exit data copyout(a[0:n])
int err = 1; //Failing
for (int x = 0; x < n; ++x){
  if (fabs(a[x] - a_copy[x]) > PRECISION){
    err = 0; //Passing
  }
}
```

In this test, we initialize the data to be random values and make a copy of the data for verification. We then create the one set of data on the device. This would initialize the data over garbage data on the device. On the following line, we copyout the data, which would copy out the garbage data on the device. After this, we iterate over the data and make sure that it is different from the copy of the initial data. If it is different, it demonstrates that the allocation happened without the copy of data.

Now, while this test is well designed to test that the data was not copied, the test fails to consider the possibility that the test could be operating on a shared memory device. If this test was running on a machine with a shared memory device, at line 7, when the data is copied in, the memory would not be allocated and any device operations would occur on the data that is also available to the host. The `exit data` on the following line would again do nothing. During the loop on lines 10–14, the loop would operate again on the data as it was initially and would fail the conditional statement within every iteration, resulting in a failing test. However, this problem could be worked around as shown in Code 6.

Code 6. Enter Data Create V.2

```
double * a = (double *)malloc(n * sizeof(double));
double * a_copy = (double *)malloc(n * sizeof(double));
int devtest = 1;
int err = 0;
#pragma acc enter data copyin(devtest)
#pragma acc parallel present(devtest)
{
  devtest = 0;
}
for (int x = 0; x < n; ++x){
  a[x] = rand();
  a_copy[x] = a[x];
}
if (devtest == 1){
  #pragma acc enter data create(a[0:n])
  #pragma acc exit data copyout(a[0:n])
  err = 1; //Failing
  for (int x = 0; x < n; ++x){
    if (fabs(a[x] - a_copy[x]) > PRECISION){
      err = 0; //Passing
    }
  }
}
```

Here, we create a new variable, ‘devtest’, as an `int` that is meant to test the presence of a separate memory device before entering into separate memory dependent code. We initiate the variable to a passing condition and copy it into the device memory. If the device memory is separate, the data transfer occurs and its value on device is subsequently updated to a failing condition. However, at the exit of the `parallel` region, since the scalar was explicitly copied in, there is no implicit copy back to the host data. Thus, the host version of the devtest variable is still in the passing condition. However in the case of a shared memory system, we have the same issue as the previous version of the test. The `parallel` region updates the host version, causing the devtest variable to be in the failing condition. This allows the test to bypass the test conditionally on the presence of a device. The new functionality of conditionally skipping separate memory dependent testing is also dependent on proper data management, which increases the chances of misdiagnosing issues.

Code 7. Enter Data Create V.3

```
double * a = (double *)malloc(n * sizeof(double));
double * a_copy = (double *)malloc(n * sizeof(double));
int * devtest = (int *)malloc(sizeof(int));
int err = 0;
devtest[0] = 1;
#pragma acc enter data copyin(devtest[0:1])
#pragma acc parallel present(devtest[0:1])
{
  devtest[0] = 0;
}
for (int x = 0; x < n; ++x){
  a[x] = rand();
  a_copy[x] = a[x];
}
if (devtest[0] == 1){
  #pragma acc enter data create(a[0:n])
  #pragma acc exit data copyout(a[0:n])
  err = 1; //Failing
  for (int x = 0; x < n; ++x){
    if (fabs(a[x] - a_copy[x]) > PRECISION){
      err = 0 //Passing
    }
  }
}
```

While Code 7 is almost exactly the same, we change the type of the devtest variable to be a `int *`. A strict reading of the specifications describes the data transfer protocols for scalar variables in parallel regions as being treated as if it appeared in a firstprivate clause for the region if the scalar has not already appeared in another data clause or in a surrounding data region, in which case, the operation is that of the explicit data clause. This means that partial implementations could potentially implement the implicit data transfer without allowing them to be overridden by other data clauses. Instead, by using a pointer to the data, it gets treated as a non-scalar, avoiding any potential issues with implicit data transfer and getting incorrect results with regards to the presence of a separate memory device.

There is yet another issue with this test. The test, which creates the data and copies it back to host, relies on the assumption that the data on device will be allocated over garbage data that has different values than that of the host version of the data. While relatively unlikely, this allows for mixed results that, if users are testing for only strict adherence of the language, may be unwelcome. Instead of removing a test over a minor risk or keeping a code that may give unreliable results, we instead move it to a conditional region dependent on configuration settings of the test-suite as seen in Code 8 on line 16.

Code 8. Enter Data Create V.4

```
double * a = (double *)malloc(n * sizeof(double));
double * b = (double *)malloc(n * sizeof(double));
double * a_copy = (double *)malloc(n *sizeof(double));
int * devtest = (int *)malloc(sizeof(int));
int err = 0;
devtest[0] = 1;
#pragma acc enter data copyin(devtest[0:1])
#pragma acc parallel present(devtest[0:1])
{
  devtest[0] = 0;
}
for (int x = 0; x < n; ++x){
  a[x] = rand();
  a_copy[x] = a[x];
}
if (devtest[0] == 1 && run_probabilistic == 1){
  #pragma acc enter data create(a[0:n])
  #pragma acc exit data copyout(a[0:n])
  err = 1; \\Failing
  for (int x = 0; x < n; ++x){
    if (fabs(a[x] - a_copy[x]) > PRECISION){
      err = 0;
    }
  }
}
for (int x = 0; x < n; ++x){
  a[x] = rand();
  b[x] = 0.0;
}
#pragma acc enter data copyin(a[0:n]) create(b[0:n])
#pragma acc parallel present(a[0:n], b[0:n])
{
  for (int x = 0; x < n; ++x){
    b[x] = a[x];
  }
}
#pragma acc exit data copyout(b[0:n]) delete(a[0:n])
for (int x = 0; x < n; ++x){
  if (fabs(a[x] - b[x]) > PRECISION){
    err += 1;
    break;
  }
}
```

In order to add at least some minimal testing for systems that either have shared memory or are being run without probabilistic tests, we also include a test that is written to be completely independent of device type or garbage data. On lines 30–43 we test the `create` clause, but instead of testing the lack of data transfer, we only test that the data was allocated and available for use. While this severely limits this test, it does provide proof of the dependability of the clause for any standard code that doesn't depend on any non-deterministic structures.

This example demonstrates some of the difficulties that come with varying memory models. While these are not inherent to a parallel programming model, many devices that run these codes depend on separate memory while other technologies, such as Knight's Landing, focus on reunifying the memory to minimize the startup cost to parallelization. There are other issues to protect the tests against in terms of the non-sequential processing model which is inherent to any parallelization.

In the case of testing the `reduction` clause on a `loop` directive with a multiply operator, one of the problems that can be run into with this sort of reduction is overflowing the reduction variable. In order to sanitize the data to avoid this, it might be tempting to keep a rolling multiplied total when initializing the data and limiting the randomization to a range that will not allow for overflow. When processing sequentially, there would be no problems with this solution. However, since the reduction does not execute sequentially, all the largest of the numbers could, by chance be multiplied together before any of the lower ones, which could cause undefined values before the completion of the reduction.

4 Setup, Compilation Flags and Infrastructure

For our tests, we ran the suite on three different systems. The first system is University of Delaware's (UD) Community Cluster, Farber,[3] where our compute node consists of Intel Xeon CPU E5-2670 v2 processor (20 cores) of Ivy Bridge architecture and a single NVIDIA Tesla K80 GPU. The second system is Titan[4] housed at ORNL where each compute node contains 1 AMD Interlagos 6274 processor (16 cores) of the Bulldozer architecture and a single NVIDIA Tesla K20 GPU. The third system is Summitdev consisting of IBM Power8+ processor CPUs (20 cores) with 4 NVIDIA Tesla P100 GPUs. We were also able to test on a machine running on an Intel Knights Landing chip, the Xeon Phi 7230.

With the first two systems, we were able to use PGI 16.10, which is also the version for the community edition (at the time of writing this paper). For the first system - UD's Farber, we use 17.3 (the latest version of the compiler at the time of writing this paper). On the third system and the Knights Landing system, we use PGI 17.1 due to the lack of availability of the 16.10 version on that system. We also used the GNU 6.3-20170303 compiler version on the second system, Titan. Between these various environments, we were able to test all supported

[3] http://docs.hpc.udel.edu/clusters/farber/start#farber.

[4] https://www.olcf.ornl.gov/titan/.

platforms for both PGI and GNU except for little-endian PowerPC systems for GNU. In all, we ran the testsuite 13 times as shown in Table 1 with variations of compiler vendor, compiler version, platform type, and platform version to ensure we do not spot any flaky tests.

Table 1. Compiler versions and platforms

Run	Compiler	Platform
1	PGI 16.10	Ivy Bridge Multicore
2	PGI 16.10	K80
3	PGI 17.3	Ivy Bridge Multicore
4	PGI 17.3	K80
5	PGI 16.10	Bulldozer Multicore
6	PGI 16.10	K20
7	PGI 17.1	Power8+ Multicore
8	PGI 17.1	P100
9	PGI 17.1	Knights Landing
10	GNU 6.3-20170303	Bulldozer Multicore
11	GNU 6.3-20170303	K20
12	GNU 6.0.0-20160415	Ivy Bridge Multicore
13	GNU 6.0.0-20160415	K80

The compilation uses various flags. For instance with the PGI compiler, to target the CPUs either the -ta=multicore or -ta=host flags is used (multicore for multithreading and host for single threaded execution). For the GPUs, the target accelerator flag is set to -ta=tesla for the Kepler cards or -ta=tesla:cc60 when targeting the P100 cards. With the GNU compiler, targeting is not handled at compile time. Instead, both host and device versions of the code are built. In order to run the host version (There is no multicore option for GNU as of yet), the internal control variable has to be set with ACC_DEVICE_TYPE=host, or, in order to force device operation, ACC_DEVICE_TYPE=nvidia.

With the GNU compiler, compilation failure due to linking issues were sometimes resolved by adding the -lm flag. If the issues were still not resolved, adding the -foffload=-lm would occasionally fix the compilation error as well. With fortran codes, due the strict adherence of GNU to fortran specifications, the -ffree-line-length-none could be used when lines were longer than 72 characters.

5 Results

(Note: We have used 16.10 wherever possible as at the time of writing this paper PGI's Community Edition supports 16.10[5]). In executing the testsuite on

[5] https://www.pgroup.com/products/community.htm.

the assortment of platforms discussed in Sect. 4, we can statistically verify the integrity of these tests. We developed 177 tests and out of the 177 tests, 89.3% passed in all runs with the PGI compiler and 51.7% passed in all runs with PGI and GNU compilers. In addition, no test failed all runs. The overall success rate for the tests is 90.6% across 2301 individual test-runs (includes the combination of compiler versions on the variety of hardware architectures).

5.1 Comparison of PGI Compiler Targeting Various Architectures

Between the systems we had access to, we were able to test seven separate architectures. As far as accelerators are concerned, we were able to run the testsuite on NVIDIA's K20 architecture, K80 architecture, and the P100 architecture. Of these, on the P100 we were not able to use PGI's 16.10 compiler version; instead we used PGI's 17.1 compiler version available on that system. However, targeting these accelerators was very uniform, having a total pass rate of 525/531 or 98.9%. Across all platforms, tests for the `tile` clause on a `parallel loop` in C and the test for the `firstprivate` clause on a `parallel` region in Fortran failed. However, the test for the `tile` clause is not a surprising failure. The test takes the lack of a range of valid values for the arguments in the specifications to test that all values should work. As an optimization clause, even if the tiling arguments are out of bounds for the loops, it should not change the results of the test. On other versions of the test that did not do this, the `tile` clause did execute properly.

Targeting multicore processors/shared memory devices had much more varied results. We were able to run the testsuite on the Intel Ivy Bridge architecture, the AMD Bulldozer architecture (TITAN), the IBM Power8+ architecture (Summitdev), and the Intel Knights Landing (KNL) architecture (though KNL is not yet officially supported by PGI). We used PGI's 16.10 version on the Ivy Bridge and Bulldozer architectures, and PGI's 17.1 version on Power8+ and Knights Landing. The average pass rate for these architectures was lower than that of the discrete graphics cards tested. Instead of 98.9% pass rate, these tests averaged a 95.3% pass rate or 675/708 passing over total, though this is not surprising, due to PGI's relatively recent support of OpenACC targeting multicore on x86 and Power processors.

Across all multicore architectures, the test for the `firstprivate` clause on a `parallel` construct in C and the test of a multiplication `reduction` in a `parallel` region in C were the only consistent failures. Others were also very close to across the board failure, such as the Fortran versions of each of these tests, which failed all but the Ivy Bridge architecture, and the test of an *AND* `reduction` on a `parallel loop` with a scalar that has been privatized in a surrounding loop in C, which failed on all but Power8+. The test of the `tile` clause on a `parallel loop` in C failed on all architectures but Bulldozer. Both the test of the *AND* `reduction` on a `parallel loop` in C and the test of the *OR* `reduction` on a `parallel loop` in C also failed for both Ivy Bridge and Knights Landing. In particular, though, the Power8+ architecture had a good amount of additional failures that should be noted. The Power8+ architecture failed a simple test having multiple loops inside of a `parallel` construct in both C and Fortran as well as many of the tests that utilized the `create` clause. The tests associated with the `create` clause that failed were the tests that used the functionality of removing the lower bound on the data clause to have it default to zero on a `data` construct in both C and Fortran and on an `enter data` directive in Fortran and the use of a `create` clause on a `parallel` construct in both C and Fortran.

The performance of these various platforms can be seen in Table 2.

Table 2. Performance of PGI architecture targeting

Architecture	Pass rate	Percent passed
K20	175/177	98.9%
K80	175/177	98.3%
P100	175/177	98.9%
Ivy Bridge	171/177	96.6%
Bulldozer	172/177	97.2%
Power8+	165/177	93.3%
Knights landing	167/177	94.4%

5.2 Comparison of Various PGI Compiler Versions

We were also able to test a series of PGI's compilers using a NVIDIA K80 as shown in Table 3 The first version that we were able to test was 14.10 (at the time of this version, PGI did not support the 2.5 specifications). Thus, there is quite a

Table 3. Comparison of K80 targeting across PGI versions

Compiler version	Fortran pass rate	C pass rate	Fortran percent passed	C percent passed
14.10	60/86	67/91	69.8%	73.6%
15.1	64/86	80/91	74.4%	87.9%
15.5	65/86	80/91	75.6%	87.9%
15.10	68/86	84/91	79.1%	92.3%
16.1	69/86	84/91	80.2%	92.3%
16.4	82/86	84/91	95.3%	92.3%
16.7	85/86	90/91	98.8%	98.9%
16.10	85/86	90/91	98.8%	98.9%
17.1	85/86	90/91	98.8%	98.9%
17.3	85/86	90/91	98.8%	98.9%

noticeable improvement across 14.10 to 15.1, causing issues with about nineteen of the tests. In the 15.5 version, the `tile` clause was added, fixing the errors associated with the Fortran version of the test of the `tile` clause, though the C version continues to have errors even now due to its inability to handle abnormal arguments for the `tile` clause. With the 15.10 version, PGI added support for the `num_gangs`, `num_workers`, and `vector_length` for the `kernels` construct, resolving compilation errors caused by the use of these clauses. Also, until the 16.4 version, having variables copied into the device multiple times would cause errors in Fortran. This situation appears in another fourteen of the tests. With 16.7, reference counting was added (including the finalize clause), fixing another five tests. Also with 16.7, the `num_gangs`, `num_workers`, and `vector_length` clauses were added to the `kernels` construct in the C language, fixing another three tests. The remaining issues have been reported to PGI in order to be fixed.

5.3 Comparison of PGI 16.10 and PGI 17.3 Multicore Support

In our runs on the UD Farber machine, we were able to compare results between the 16.10 (community edition) and the 17.3 (latest edition) of the PGI compiler on both Ivy Bridge multicore and NVIDIA K80. Overall the success rate of the 16.10 version was 346/354 between both platforms. Of the seven failures on the 16.10 version targeting the Ivy Bridge multicore, four were failures during testing the reduction clause in the C language. However, in the 17.3 version, all but one of these is resolved. Also, when targeting K80, there are no such failures. There also seem to be issues working with the `firstprivate` clause. With both version, Ivy Bridge failed the test of the `firstprivate` clause in the C language while on K80, both versions failed the Fortran version. Also, both versions exhibited some shortcomings in dealing with `async` clauses on `parallel` regions when targeting K80. The last issue is with the `tile` clause. While the operation seems to be proper in many cases, there seems to be an issue with properly tiling when the tiling arguments fall beyond the bound of the iterations in the nested loops. For these specific platforms, we only see an improvement with the multicore targeting when using the `reduction` clause, bringing the 17.3 version's success rate to 349/354. It is possible, though, that testing in a similar fashion on other platforms would show a greater/other improvements that are not evident here.

5.4 The GNU Compiler's Accuracy to the OpenACC Specifications

We also were able to use the GNU compiler on the Farber system and Titan. While it is far from competitive with the PGI results, GNU also does not, at this point, support the features that were added in the 2.5 version of the OpenACC specifications. Notably among these new features are reference counting and the new ways memory is managed. Instead of using `pcopy` or `copy`, now the functionality of `copy` has been completely removed in favor of treating all `copy` clauses as `pcopy`. This drastically changes the way the tests are interpreted by the compiler. Many of the tests use some form of multiple references in order to test proper data management and thus the GNU compiler is, by supporting the

2.0 version, predestined to fail. On the Farber machine, 38 of the tests failures were memory related runtime errors. While we cannot guarantee that these errors would be fixed with support for 2.5, many most likely would pass. Results for GNU in Table 4 indicate ACC_DEVICE_TYPE is set to host; single-threaded host-fallback execution, in a shared-memory mode. An in-depth evaluation of GCC OpenACC implementation on Cray systems is discussed in[6]. However these discussions are based on an older version of the testsuite.

Table 4. GNU vs. PGI pass rates

Architecture	PGI pass rate	GNU pass rate
K20	175/177	112/177
K80	175/177	113/177
Ivy Bridge	171/177	154/177
Bulldozer	172/177	157/177

6 Discussion

With this testsuite, we have shown both the status of the compilers and their ability to target various architectures. We also had the opportunity to run the suite on Cray compiler. However, due to the compiler's adherence to an outdated version of the OpenACC specifications, and lack of demand from users to use Cray OpenACC compiler we have not summarized our test results on the compiler. Also, since we plan to release the testsuite, anyone interested to validate Cray OpenACC implementations is welcome to use our testsuite to validate Cray's OpenACC implementation.

When using the `firstprivate` clause in the Fortran language using the PGI compiler, due to it's potential for causing errors we recommend starting debugging there. One potential solution to if it will not work is to replace the `firstprivate` clause with a `private` clause and initialize the data in a gang redundant loop. Though this will take it's toll on performance, it could solve incorrect calculations or runtime errors.

Our testsuite has helped validate OpenACC compilers on the Titan supercomputer and the pre-exascale machine Summitdev at Oak Ridge National Laboratory (ORNL). Our tests have already been integrated to the official harness testsuite of Titan.

There are multiple types of users with quite different requirements for a comprehensive validation suite, the accessibility and usability features need to be flexible. End users need an easy way to run all of the tests in the suite and see a useful summary of the results to know how much coverage their software stack and architecture supports. However, when using for QA purposes,

[6] https://cug.org/proceedings/cug2017_proceedings/includes/files/pap174s2-file1.pdf.

an implementer needs to be able to easily run and debug at the single-test level of granularity. These different types of requirements require a robust testsuite infrastructure that is currently not in place but under discussion. This infrastructure needs to be simple so that anybody can contribute new tests or fix bugs by only needing to understand the host language and OpenACC, not the details of the test harness. Only by minimizing the effort for new contributions will the test be widely adopted and expanded by the community.

7 Related Work

Our first paper on this effort [24] covers the specifications through version 1.0 of the OpenACC specifications. A closely related work to this effort is the OpenMP Validation Suite [20] that also validates and verifies the features of OpenMP compiler implementations. In 2003, an OpenMP validation suite was developed [12] to validate OpenMP implementations for OpenMP 2.0. This work was extended [13] to build test cases to validate implementations of OpenMP 2.5 features. This work was further extended [23] to develop a more robust OpenMP validation suite and provided up-to-date test cases covering all the features until OpenMP 3.1.

Other related efforts to building and using 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 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 [10] 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 programs tests are referred to as the "LLVM test suite" (or "test-suite"). The tests itself are driven by *lit* testing tool, which is part of LLVM.

The parallel testsuite [7] 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 compare the ability of different Fortran compilers to automatically parallelize various loops. The Parallel Loops test suite is modeled after the Livermore Fortran kernels [11].

8 Conclusion and Future Work

This project develops test cases to validate and verify compilers' implementations of OpenACC features as of Version 2.5. As the features of the programming model have evolved, so has the testsuite. The tests have enabled identification of compiler bugs that have been or are being fixed in subsequent compiler versions, thus improving the quality of the compilers. In addition to testing the platforms and compilers with the testsuite as shown in the results, the variety of compiler environments and hardware platforms have evaluated the tests to verify that they properly conform to OpenACC specification.

We aim to build a comprehensive OpenACC testsuite for conformance of the language features in the OpenACC specification. To that end, we are adding tests to cover corner cases that may otherwise be not possible via simple unit tests. We will also add tests to cover features as they are added to the specification. We will also build interpreters to generate for each test a variety of variations on that test to test fringe cases and feature limitations such as testing each numeric type for each operator in a given parallelized region or testing limitations on optimization variables. To make the testsuite easily usable, we will create forward and backward references for the testsuite with the specification such that each test in the open-source GitHub repository can be related to a definition in the specification and definitions in the specification can be tagged to a test in the repository.[7]

Acknowledgments. We are very grateful to OpenACC and NVIDIA for supporting this project. Special thanks to Mathew Colgrove, Duncan Poole, Christophe Harle, Jeff Larkin, Michael Wolfe, James Beyer, Pat Brooks, Doug Holt, Wael Elwasif, Thomas Swinge, Cesar Philippidis, Randy Allen and Alex Rech.

This material is based upon work supported by the U.S. Department of Energy, Office of science, and this research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

1. CAAR center for accelerated application readiness. https://www.olcf.ornl.gov/caar/
2. INCITE program. http://www.doeleadershipcomputing.org/incite-program/
3. NVIDIA Thrust. https://developer.nvidia.com/thrust. Accessed 03 Feb 2017
4. OpenACC
5. Adams, M.F., Ethier, S., Wichmann, N.: Performance of particle in cell methods on highly concurrent computational architectures. J. Phys. Conf. Ser. **78**(1), 012001 (2007)
6. Aidas, K., Angeli, C., Bak, K.L., et al.: The dalton quantum chemistry program system. Wiley Interdiscip. Rev. Comput. Mol. Sci. **4**(3), 269–284 (2014)
7. Dongarra, J., Furtney, M., Reinhardt, S., Russell, J.: Parallel loops–a test suite for parallelizing compilers: description and example results. Parallel Comput. **17**(10–11), 1247–1255 (1991)
8. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014)
9. Ethier, S., Tang, W.M., Walkup, R., Oliker, L.: Large-scale gyrokinetic particle simulation of microturbulence in magnetically confined fusion plasmas. IBM J. Res. Dev. **52**(1.2), 105–115 (2008)

[7] For more detailed explanation and example, see https://github.com/OpenACCUserGroup/OpenACCV-V/blob/master/README.md.

10. LLVM. Llvm Testing Infrastructure Guide. http://www.llvm.org/pre-releases/4.0.0/rc2/docs/TestingGuide.html#test-suite
11. McMahon, F.H.: The livermore fortran kernels: a computer test of the numerical performance range. Technical report, Lawrence Livermore National Lab, CA, USA (1986)
12. Müller, M., Neytchev, P.: An openMP validation suite. In: Fifth European Workshop on OpenMP, Aachen University, Germany (2003)
13. Müller, M., Niethammer, C., Chapman, B., Wen, Y., Liu, Z.: Validating openMP 2.5 for fortran and C/C. In: Sixth European Workshop on OpenMP, KTH Royal Institute of Technology. Citeseer (2004)
14. Nonaka, A., Almgren, A.S., Bell, J.B., Lijewski, M.J., Malone, C.M., Zingale, M.: MAESTRO: an adaptive low mach number hydrodynamics algorithm for stellar flows. Astrophys. J. Suppl. Ser. **188**(2), 358 (2010)
15. NVIDIA. CUDA SDK Code Samples. http://developer.nvidia.com/cuda-cc-sdk-code-samples. Accessed 03 Feb 2017
16. OpenACC. Deep Copy Attach and Detach. http://www.openacc.org/sites/default/files/TR-16-1.pdf
17. OpenACC. OpenACC, Directives for Accelerators. http://www.openacc.org/
18. OpenCL. OpenCL. https://www.khronos.org/
19. OpenMP. OpenMP 4.5 specification. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
20. OpenMP Validation and Verification Suite. OpenMP 3.1 Specification. https://github.com/sunitachandra/omp-validation
21. Taylor, M.A., Edwards, J., Cyr, A.S.: Petascale atmospheric models for the community climate system model: new developments and evaluation of scalable dynamical cores. J. Phys. Conf. Ser. **125**(1), 012023 (2008)
22. Taylor, M.A., Edwards, J., Thomas, S., Nair, R.: A mass and energy conserving spectral element atmospheric dynamical core on the cubed-sphere grid. J. Phys. Conf. Ser. **78**(1), 012074 (2007)
23. Wang, C., Chandrasekaran, S., Chapman, B.: An OpenMP 3.1 validation testsuite. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 237–249. Springer, Heidelberg (2012). doi:10.1007/978-3-642-30961-8_18
24. Wang, C., Xu, R., Chandrasekaran, S., Chapman, B., Hernandez, O.: A validation testsuite for OpenACC 1.0. In: 2014 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1407–1416. IEEE (2014)
25. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: ACM SIGPLAN Notices, vol. 46, pp. 283–294. ACM (2011)