2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

A Validation Testsuite for OpenACC 1.0

Cheng Wang[†*], Rengan Xu[†*], Sunita Chandrasekaran[†], Barbara Chapman[†], Oscar Hernandez[‡]
[†]Department of Computer Science, University of Houston, Houston, TX, USA
Email: {cwang35,uhxrg,sunita,chapman}@cs.uh.edu
[‡]Computer Science and Mathematics Division, Oak Ridge National Laboratory
Email: oscar@ornl.gov

***Abstract*—Directive-based programming models provide high-level of abstraction thus hiding complex low-level details of the underlying hardware from the programmer. One such model is OpenACC that is also a portable programming model allowing programmers to write applications that offload portions of work from a host CPU to an attached accelerator (GPU or a similar device). The model is gaining popularity and being used for accelerating many types of applications, ranging from molecular dynamics codes to particle physics models. It is critical to evaluate the correctness of the OpenACC implementations and determine its conformance to the specification. In this paper, we present a robust and scalable testing infrastructure that serves this purpose. We worked very closely with three main vendors that offer compiler support for OpenACC and assisted them in identifying and resolving compiler bugs helping them improve the quality of their compilers. The testsuite also aims to identify and resolve ambiguities within the OpenACC specification. This testsuite has been integrated into the harness infrastructure of the TITAN machine at Oak Ridge National Lab and is being used for production. The testsuite consists of test cases for all the directives and clauses of OpenACC, both for C and Fortran languages. The testsuite discussed in this paper focuses on the OpenACC 1.0 feature set. The framework of the testsuite is robust enough to create test cases for 2.0 and future releases. This work is in progress.**

***Keywords*-Validation; OpenACC; Compiler**

I. Introduction

Recent years have seen a rise of massively-parallel supercomputing systems that are based on heterogeneous architectures combining multicore CPUs with accelerators, such as General-Purpose Graphic Processing Units (GPGPUs), Accelerated Processing Units (APUs), and Many Integrated Cores (MIC). While such systems offer a promising performance with reasonable power consumption, programming accelerators in an efficient manner is still a challenge. The existing low-level APIs such as CUDA and OpenCL usually require users to be expert programmers and restructure the code largely. Optimized kernels are written that are usually coupled with specific devices. This leads to a less productive and more error prone software development process that is challenging to be adopted by the rapidly growing HPC market.

Recent approaches to program accelerators include directive-based, high-level programming models for accelerators. It allows the users to insert non-executable pragmas and guide the compiler to handle low-level complexities of the system. The major advantage of the directive-based approach is that it offers a high-level programming abstraction thus simplifying the code maintenance and improving productivity. There are a number of related efforts that includes PGI Accelerator [1], HMPP directives [2], hiCUDA [3], and so on. As they offer different feature sets, the code portability therefore becomes a major issue. As a joint standardization between CAPS, CRAY, PGI and NVIDIA, OpenACC [4] was first released in November 2011, which aims to provide a directive-based portable programming model for accelerators. By using OpenACC, it allows the users to maintain a single code base that is compatible with various compilers, while on the other hand, the code is also portable across different possible types of platforms.

Different compiler developers may interpret a given specification differently leading to more than one way of implementing a given construct. Ambiguities in the specification could be one of the reasons. Another reason could be that the current implementation is based off of an implementation of a similar construct's functionality, yet not identical. It is quite common to encounter these circumstances especially when the specification and implementation of a standard is evolving.

As a motivational example, the OpenACC execution model defines `gang/worker/vector` clauses to specify different levels of parallelism on accelerators. They usually occur in a specific order and choosing different values in the clause will have a significant impact on the application's performance. For instance, they may correspond to the `block/warp/threads` respectively from the perspective of the CUDA model. However, the mapping is totally implementation-dependent and can give rise to various possible combinations.

For instance, Figure 1 shows a code snippet pointing out an ambiguity in the OpenACC 1.0 specification. A `worker` loop should usually occur inside a `gang` loop. But can we allow a `worker` loop without an outer `gang` loop? For instance, if the number of `gangs` is known to be equal to 1, then there is no need for a `gang` loop. In addition, if the purpose of the `worker` loop is to initialize data that is local to the `gang`, the outer `gang` loop is not needed. The specification does not state if this option is acceptable or otherwise. Hence upon executing this option, we observed different results generated by different compilers leading to inconsistency in compiler behaviors. This

[*]Equal contribution by the first two authors.

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.158

```
#pragma acc parallel
{
  #pragma acc loop worker
  for(i=0; i<N; i++) {
    ...
  }
}
```

Fig. 1. An example of ambiguity in the OpenACC specification: can we allow a `worker` loop without an outer `gang` loop?

also led to wider performance gaps while comparing different compiler results.

In this paper we propose an OpenACC compiler validation testsuite and infrastructure that is used to validate and verify different OpenACC compiler implementations for conformance, correctness and completeness. The testsuite consists of two parts: test codes and test infrastructure. We have ensured that the design of each of the test case is unique. To create an unique interpretation, single generated test code must test for only one OpenACC feature (e.g. `async`). The test code is written based on template, i.e., a test code is written following an `html` syntax structure that includes the OpenACC directive/clause to be tested. The test infrastructure written by the `perl` script will then be used to parse the template and automatically generate the associated test codes. The use of template-based test has several advantages. First of all, it only needs minimum efforts to develop the completed test code. As a result, developers only need to focus on designing the test cases instead of redundantly writing the entire test programs including the main function and input/output every time. The generated test code is a complete and standalone C/Fortran code, i.e., it could be compiled by any OpenACC compiler. A test harness will then compile the program, run the executable, check for the results and generate reports. To provide a confidence level for its correctness, the test infrastructure also generates a set of corresponding cross-test programs along with the feature tests. A bug report is generated that qualitatively and quantitatively analyzes features that are being tested. The infrastructure is extensible enough to accommodate newer features of the specification as and when the standard evolves. This testsuite has been integrated into the harness infrastructure of the TITAN machine at Oak Ridge National Lab and is being used for production.

The tests are generated in the form of a tree structure: it begins by covering OpenACC directives followed by clauses belonging to those directives, as well as the runtime routines and environment variables. Currently it covers the OpenACC 1.0 feature set. Although this paper only focuses on the tests for OpenACC 1.0, at the time of writing, OpenACC 2.0 was released. We are in the process of writing test codes for the newer features added to 2.0.

In order to make this effort successful, we have been collaborating with vendors, NVIDIA/PGI, CAPS and Cray since the inception of the standard. We work very closely with the vendor team. We identify and report bugs found in their OpenACC implementations. The vendors fix them and inform us when a newer version of the compiler is released. We then verify if the issues were resolved. We receive feedback not only from the vendors but also from the users. The benefit here is two-fold. The feedback helps us to improve the quality of the test cases and at the same time validate more thoroughly conformance of the features to the standard. We observed that bugs resulted typically from either implementing certain complex directives such as `reduction` or due to misinterpretation of the specification.

This paper is written in a manner that we do not simply report about the bugs but we also share our experiences in creating and constructing test cases. We also discuss about the design of a robust infrastructure and the challenges that came along the way. The contributions made in this paper with joint effort from vendors and users are:

Primarily our testsuite is able to assist compiler developers to validate their compiler's conformance to the specification. In addition, our testsuite is especially important to OpenACC users as we observed that for most of the time compilers just emit *wrong code bugs*: the bugs that cause compilers to generate wrong results in silence without a verbose alarming information. Secondly, at times results generated from different compilers differ from each other, these are not due to the bugs in the compilers, but due to the ambiguities in the specification or in other words different interpretations of the specification by different compiler developers. This is quite the case when the specification is relatively new and evolving. Our aim was to capture such cases during the process. To our delight, most of the ambiguities we reported were resolved in OpenACC 2.0.

The rest of paper is organized as follows: Section II briefly gives an overview of the OpenACC programming model and the feature set. In section III, we explain the design of the validation testsuite. We demonstrate several typical test cases in Section IV. Section V elaborates how our validation suite evaluates the OpenACC compilers. Section VII shows our suite being currently used to validate the functionality of the programming environment of Titan. In Section VIII we discuss some of the related works to this effort. Section IX concludes our work.

II. Overview of OpenACC Programming Model

OpenACC is an emerging standard for programming accelerator boards in conjunction with a host CPU, which could be a multicore platform. It is based on the use of `pragmas` or `directives` that allow the application developers to mark regions of code for acceleration in a vendor-neutral manner. It builds on top of prior efforts by several vendors (notably PGI and CAPS Enterprise) to provide parallel programming interface for heterogeneous systems, with a particular emphasis on platforms that are comprised of multicore processors as well as GPUs. Among others, OpenACC is intended for use on the nodes of large-scale platforms such as the Titan system at ORNL, where CPUs and NVIDIA GPUs are used

in concert to solve some of the nations most urgent scientific problems.

Recently OpenACC 2.0 [5] was ratified. Late last year, Cray has introduced new OpenACC 2.0 support for directives-based programming of accelerators and coprocessors in Cray supercomputers [6]. PGI and CAPS will be soon introducing support for 2.0 in their compilers.

The OpenACC feature set includes pragmas, or directives, that can be used in conjunction with C, C++ and Fortran code to program accelerator boards. OpenACC can work with OpenMP to provide a portable programming interface that addresses the parallelism in a shared memory multicore system as well as accelerators. A key element of the interface is the parallel construct that launches gangs that will execute in parallel. Each of the gangs may support multiple workers that execute vector or SIMD constructs. A variety of clauses are provided that enables conditional execution, controls the number of threads, specifies the scope of the data accessed in the accelerator parallel region and determines if the host CPU should wait for the region to complete before proceeding with other work.

Suitable placement of data and careful management of required data transfer between host and accelerator is critical for application performance on the emerging heterogeneous platforms. Accordingly, there are a variety of features in OpenACC that enables the application developer to allocate data and determine whether data needs to be transferred between the configured devices. The features also enable control this transfer, including the values to be updated on the host/accelerator by copying current data values on the accelerator/host, respectively. These features are complemented by a set of library routines to obtain device information or set device types, test for completion of asynchronous activities, as well as a few environment variables to identify the devices that will be used.

OpenACC standard gives great flexibility to the compiler implementation. For instance, different compilers can have different interpretation of OpenACC three level parallelism: coarse grain parallelism "gang", fine grain parallelism "worker" and vector parallelism "vector". On an NVIDIA GPU, PGI maps each gang to a thread block, and vector to threads in a block and it just ignores worker; CAPS maps gang to the x-dimension of a grid block, worker to the y-dimension of a thread block, and vector to the x-dimension of a thread block; Cray maps each gang to a thread block, worker to warp and vector to SIMT group of threads.

III. The Test Infrastructure Design

In this section, we discuss the design and implementation about our testsuite infrastructure. The primary goal of our validation suite is to provide a set of short feature tests wherever possible and check if the directive and the clauses associated that are being tested have been implemented correctly. For example, we test the `parallel` construct and its corresponding clauses (e.g. `async`, `num gangs`, `private`, `firstprivate`, and so on).

The testsuite will check if the directive passed or failed by verifying the result with a pre-calculated value. If the values do not match, it implies that there is an implementation issue. We define these tests as *functional tests*. An important point to note is that a false positive can be an output with the functional tests. This is because there might be more than one directive that is being used at a given point of time.

For instance, figure 2(a) shows a simplified test case for `loop` directive. It partitions and assigns the loop iterations to the number of `gangs` specified in the `parallel` construct. So in this case each element of the array `A` is incremented exactly once. Since the `loop` directive must be used within the `parallel` construct, while the functional test for the `loop` feature may pass with an expected result, that may simply be due to the use of the `parallel` construct. To

```
#pragma acc parallel num_gangs(10)
{
  #pragma acc loop
  for(i=0; i<n; i++)
    A[i] = A[i] + 1;
}
```

(a) Functional test

```
#pragma acc parallel num_gangs(10)
{
  // loop directive is removed
  for(i=0; i<n; i++)
    A[i] = A[i] + 1;
}
```

(b) Cross test

Fig. 2. Test case for the `loop` directive

gain more confidence of the test result, we designed a deeper test methodology, namely *cross test*. This test will help in validating "only" the directive under consideration. The basic idea is that if we remove the directive being tested from the test code, the cross test should yield an "incorrect" result. For instance, in the figure 2(b), `#pragma acc loop` is simply removed for the cross test. As a result, each element should be incremented by each gang (so in total 10 times). On the other hand, if the result in the cross test is still the same as the functional test, it indicates that the directive being tested does not take any effect. The result will be reported and the functional test will be re-designed. In some instances, simply by removing the directive being tested will not work. We intentionally replace the directive being tested with another one. For example, we can help validate `firstprivate` clause by replacing `firstprivate` with a `private` clause and check the impact on the result.

The test results are statistically analyzed and each test is repeated multiple times. In order to estimate the probability that a test passes accidentally we take the following approach: if n_f is the number of failed cross tests and M the total number of iterations, the probability that the test will fail is $p = \frac{n_f}{M}$. Thus the probability that an incorrect implementation passes the test is $p_a = (1-p)^M$, and the certainty of test is $p_c = 1 - p_a$, i.e. the probability that a directive is validated. While evaluating, if the probability is 100%, we conclude that the test passed.

In the current OpenACC validation testsuite, we have designed more than 160 test cases covering the OpenACC C and OpenACC Fortran feature set included in 1.0 version. These test cases cover tests for directives, clauses, runtime library routine, as well as environment variables. Each test has two versions: functional test and cross test. We observed that one of the challenges in constructing test cases for this scenario is that if we implement each of these tests separately, the entire testsuite will become ad-hoc, error-prone, and difficult to maintain.

To improve the usability and extensibility, we also created a test infrastructure to automate the test generation as well as collect and analyze test results in qualitative and quantitative manner. Figure 3 shows the framework of the test infrastructure. A test template is written following an `html` syntax structure that includes the OpenACC directive/clause to be tested. A `perl` script is used to parse the template and automatically generate the associated test codes for both functional and cross tests. These generated test codes are C or Fortran programs that can be compilable by any OpenACC C and Fortran compilers. As discussed above, first we perform the functional test. If the feature passes the test, the feature will need to undergo a deeper test, i.e. the cross test. If the feature did not pass the functional test, a "failure" will be directly reported to the result analyzer bypassing the necessity to do the cross test. One of the advantages of using template-based testing is that only one test base is needed for each of the OpenACC features being validated and the infrastructure will automatically generate the different test programs. In addition, we only need to focus on developing the test cases instead of redundantly writing the entire test codes every time. The major features of the validation suite are as follows:

- **Compiler configuration**: User can set the configuration details for the compiler implementation to be validated.
- **Feature selection**: User can choose to test the directives, their clauses or any other feature of their choice at a given point of time.
- **Extensible test infrastructure**: We implemented two types of tests (i.e. functional and cross tests) for each of the features defined in the specification. We generate the source code by parsing the test template, compiling and executing it by using the compiler whose implementation needs to be validated. If it passes the functional test successfully, we move to the next stage, i.e. execute the cross test. In the event that the test fails to pass through the functional test, a report is generated about the test's failure.
- **Results**: After conducting the tests, based on the statistical analysis discussed earlier, user is informed about the tests that passed. For the tests that failed, our infrastructure provides the possible reasons of failure such as compilation error, incorrect results, time out and so on. After all the tests are executed, a full report will be generated demonstrating the result for each of the features. We append the bug reports with code snippets for vendors' convenience. We can generate the validation results in any of the formats such as plain text, HTML and CSV.

```
gangs_red = (int*)malloc(gangs*sizeof(int));
for(i=0; i<gangs; i++)
   gangs_red[i] = 0;

#pragma acc parallel copy(gangs_red[0:gangs]) \
                     num_gangs(gangs) \
                     num_workers(workers)
{
    #pragma acc loop gang
    for(i=0; i<gangs; i++){
        int to_reduct = 0;
        #pragma acc loop worker reduction(+:to_reduct)
        for(j=0; j<workers_load; j++)
            to_reduct++;
        gangs_red[i] = to_reduct;
    }
}

error = 0;
for(i=0; i<gangs; i++){
    if(gangs_red[i] != workers_load)
        error++;
}
return (error == 0);
```

Fig. 4. Test case for `parallel num_workers`

IV. Design of Test Cases

In this section, we discuss the design ideas of the test cases written. (Due to space constraints, we will discuss the design ideas of a few of the test cases. The rest of the tests mostly follows suit.)

A. Parallel and Kernels Construct

1) num_gangs and num_workers: The OpenACC `num_gangs` and `num_workers` define the number of `gangs` and `workers` that will execute the `parallel/kernels` region. Certain compilers may issue a compilation error, if the value that is used in `num_gangs` or `num_workers` is not accessible at the compilation time. Therefore we use a constant value for our validation test purposes. To test `num_gangs`, we use a variable with an initial value of 0 and perform reduction by all gangs, finally we compare the value after reduction with the preset value in `num_gangs`, to check if the value was equal or not, i.e. if the test is passed or failed. To test the `num_workers` clause, we set the number of gangs and workers for each gang. A two-level nested loop is designed, in which the outer loop is scheduled on all gangs and the inner loop is scheduled on all workers of one gang as shown in Figure 4. The worker-level loop performs a reduction, after which we check for the correct value of reduction at every gang.

2) private: The `private` clause is tested by creating a two dimensional matrix. Each gang writes data to each row of the matrix and all workers of one gang writes data to all the columns of that row. The starting row of the matrix is to write a private variable to each gang. There is no way to get the id of each gang. The private value of the variable will be

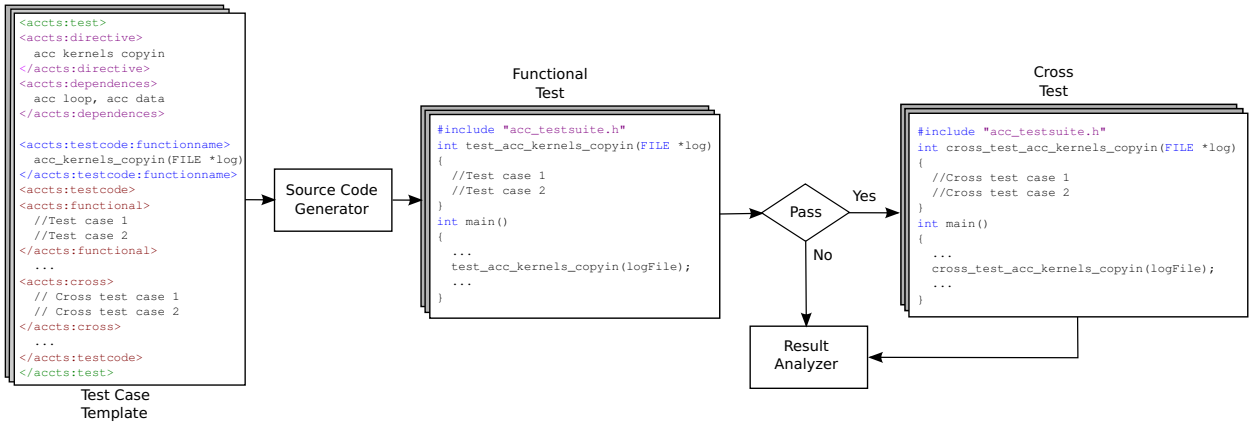


Fig. 3. Framework of the test infrastructure (in an example of `kernels copyin`)

```
N = 1000;
#pragma acc data copy(C[0:N]) copyin(A[0:N], B[0:N])
{
    sum = 1;
    for(int m=0; m<N; m++){
        #pragma acc parallel loop if (sum < N)
        for(int j=0; j<N; j++){
            C[j] += A[j] + B[j];
        }
        sum += m;
    }
}
for(i=0; i<N; i++){
    if(C[i] != 46*(A[i] + B[i]))
        error++;
}
return (error == 0);
```

Fig. 5. Test case for `parallel if`

the same for all gangs. On the host side, we check whether only the rows starting from that row are written and the values written are correct.

3) if: The test for the `if` clause is shown in Figure 5. In our test case, since the expression in the "if" clause may change at runtime, it is hard to determine the value at the compilation time. As shown in the code, N equals 1000, the sum is the reduction of m. The maximum value for m that satisfies the "if" condition is 44. At runtime, when the "if" condition is evaluated to false, the parallel region will stop running on the device. At the time when the parallel region stops running, the outer loop runs 46 times on the device.

B. Data Construct

Since most of the accelerators have discrete memory, the data movement between the host and the device is very important. The `data` construct can either be used as a standalone construct, or could be combined with `parallel` or `kernels` construct. Therefore we need to write test cases for all possible combinations. One of the clauses that this construct uses is the `if` clause, "if" condition being true, we can assume that all data copy operations are occurring. To validate if this is the case, we perform cross test where the "if" condition is false and then check if the result is different to that of the "if" condition when it was actually true.

1) data copy: The test case of `copy` clause is shown in Figure 6. The flag is initially set to HOST. This test checks for two things: first the array *A* and *B* should be copied to the device memory and array *C* copied to the host memory. Secondly, in case the code block inside the parallel region is executed on the host only, then the value of the flag will change to DEVICE, leading to an incorrect result. This is the functional test. To perform cross test, the flag is set to DEVICE unlike for the functional test where we set the flag to HOST.

2) data copyin: To test `data copyin`, we create some arrays with initial values and then copy them to the device. These arrays will do some mathematical operations inside the device to destroy the original initial values, but they will not be copied back to the host. We check the values of these arrays on the host after the operations are performed on the device or accelerator region. The values of these arrays should be exactly the same as the initial values, if not we conclude that there is an issue with the implementation of `copyin` clause.

3) data copyout: To test `data copyout`, we use two tests. The first test assigns values to a device copy array and then copies out the array to the host memory. We then check the values on the host to check if the data has been really transferred. The second test does nothing to the device copy array but we still copy it out to the host. As a result, the array values are non-deterministic because the device had just allocated memory to that array but didn't assign any values to it. Then we compare the actual undetermined values with the host's initially determined values, in order to check if they are really inconsistent.

4) data create and present-related clauses: To avoid excessive data transfer some data needs to be allowed to reside on the device. The `present` clause tells the implementation that on a non-shared memory device, the variables or arrays in the *var-list* are already present in the device memory and data need not be transferred to the device again. `Data create` can perform some operations on the data array that is only on the device side. The data is neither copied in nor copied out. After

```
flag= HOST;
for(i=0; i<N; i++){
    A[i]=i; B[i]=i;
    known_C[i]=A[i]+B[i]+DEVICE;
}
#pragma acc data create(flag) copy(A[0:N],B[0:N],C[0:N])
{
  #pragma acc parallel
  {
    flag = DEVICE;
    #pragma acc loop
    for(j=0; j<N; j++)
      C[j] = A[j]+B[j]+flag;
  }
}
for(i=0; i<N; i++){
    if((C[i]!=known_C[i]) || (flag!=HOST))
        error++;
}
return (error==0);
```

Fig. 6. Test case for `data copy`

the computations on the accelerator region, the array value should still be the same as the initial value on the host even if the value has changed during computation on the device. We consider two accelerator regions (a)`parallel` or (b)`kernel` region. In the first region we initialize data using `create` clause and we reuse the same in the next parallel region using the `present` clause. Finally we check if the final data values are the same as the pre-calculated values to verify the correctness of the implementation of `present` clause. The `present` clause can also be combined with other data clauses like `copy`, `copyin` and `copyout` and the corresponding clauses are `pcopy`, `pcopyin` and `pcopyout`. The idea of our design is to be able to validate clauses that can be used in combination.

5) data deviceptr: The `deviceptr` clause means that the pointers specified in its list are device pointers, so the data need not to be allocated or moved between the host and the device for this pointer. The test for this clause uses the runtime library routine *acc_malloc()*. The pointer returned by this routine is declared as `deviceptr` and we perform some computations to the data pointed by this pointer. Finally we copy out the data from this pointer to the host and verify its values. *acc_free()* frees the memory of the data.

C. Loop Construct

1) independent: The `loop independent` clause tells the compiler that all the iterations of the loop under consideration are data-independent with respect to each other. To validate this, we write a loop that consists of loop-carried dependencies among all iterations, i.e. each iteration depends on its previous iteration. If we use the `independent` clause, the result should be incorrect thus informing the compiler that there are dependencies within the loop. We also write a test case to check for the case when the loop is really independent.

2) seq: The clause `seq` indicates that the following loop will be executed in sequence. We use two variables *last_i* and *is_larger* to test this clause. *last_i* records the previous iteration number and *is_larger* checks whether the current iteration number is larger than the previous one. In our test, the loop increments by 1, so in each iteration we evaluate the statement $is\_larger = ((i - last\_i) == 1)\ \&\ \&is\_larger$. The initial value of *is_larger* is 1 which means it is true. After finishing all the iterations, *is_larger* is copied to the host to check if the value is still 1.

3) collapse: The `collapse` clause is used to specify how many tightly nested loops are associated with the *loop* construct. If the *collapse* clause is not present, only the immediately following loop is associated with the loop directive. Our test uses a two-level tightly nested loop and applies the `loop collapse(2) seq` before the outer loop. Inside the inner loop, the test approach of `seq` clause is used to verify that all the iterations are executed sequentially.

4) reduction: The reduction test covers combinations of different types of data (e.g. int, float and double) and different types of reduction operations (+, *, max, min, &&, ||, &, |, ^). Figure 7 shows the test case for floating-point addition reduction, which is tested by calculating $\sum_{i=0}^{N-1} ft^i$ and comparing it with the known result $(1 - ft^N)/(1 - ft)$. When comparing two different floating-point values, they are considered to be the same as long as their difference is less than a rounding error $(1.0E - 9)$.

```
N = 20; fsum = 0;
ft = 0.5; fpt = 1;
frounding_error = 1.E-9;
for(int i=0; i<N; i++){
    fpt *= ft;
}
fknown_sum = (1-fpt)/(1-ft);

#pragma acc kernels loop reduction(+:fsum)
for (i=0; i<N; i++)
    fsum += powf(ft,i);

if(fabsf(fsum-fknown_sum) > frounding_error)
    error++;
return (error == 0);
```

Fig. 7. Test for `loop reduction` addition operation for float data type

D. Update Construct

OpenACC `update` construct provides a mechanism to synchronize the device copy and host copy data at a specific location inside the data region. It includes `update host` and `update device` directives. For the `update host` test, some data is transferred to the device using `copyin` clause. Calculations are performed and data is transferred back to the host using the `update host` directive rather than `copyout` clause. On the host side we check for the correctness of this data. For the `update device` test, we use `update device` and `copyout` for the same data to verify if this data is correctly copied to the device or not. We also use `copyin` and `update device` for the same data to verify if the data is only on the device and cannot be copied back to the host.

E. Host Data Construct

The `host_data` construct makes the address of the device data available on the host. This construct has only one clause

called use_device. This clause tells the compiler to use the device address of data available in the host code. We can use some optimized procedures written in a low-level API (e.g. CUDA) by using host_data. We combine the host_data with the deviceptr test since host_data provides a device pointer while deviceptr provides a way to use it. Inside the host_data construct we call a function whose parameter is specified by the use_device clause. In the called function, some calculations are performed on the data. Finally we use the copyout clause to copy the data to the host and verify its correctness.

V. Evaluation

Our test bed is a heterogeneous system consisting of 16 cores Intel Xeon x86_64 CPU with 32GB main memory, and an NVIDIA Kepler GPU card (K20). Before we get into the details of the evaluation process, it is essential to distinguish between errors that manifest at compile time and those that happen at runtime. The *compile-time errors* are assertion violations or other internal compilation errors. For instance, this can happen if the user uses an OpenACC feature that is not yet supported by the compiler. The compile-time error can be easily captured because the compilation process will terminate and will also fail to generate the executable files. The *runtime errors* include the generation of an incorrect result; a code crash or if the code executes forever. Those errors are more vicious since most of the time, the programmers are unaware that the compilers are generating incorrect results.

A. Quantitative Comparison of CAPS, PGI and Cray Versions

Figures 8(a), 8(b), 8(c) shows plots of both C and Fortran OpenACC compilers along with number of bugs discovered in each compiler version. We tabulate the results in the Table I that shows number of bugs identified in different versions of each compiler. We notice that the number of bugs somewhat decreased with every newer version of the compiler released demonstrating improved compiler quality.

Figure 8(a) shows the plots of how CAPS compiler evolved over a period of time. We see that the pass rates for CAPS 3.0.x and CAPS 3.1.x are much lower than 3.2.x and 3.3.x versions. This is because versions 3.0.x were beta versions that were not released for public use. CAPS 3.1.x shows a lower pass rate since the declare directives had not passed the test scenarios. Probably due to priority given to other important directives such as data, kernels, loop, parallel and update. Moreover one could simply use data directives could be used instead of declare directives. Figure 8(b) shows the bugs discovered with PGI's OpenACC compiler and how they were rectified over a period of time. PGI began to provide support for OpenACC from version 12.6 onwards. We see that version 12.8 onwards shows better quality. The pass rate in 13.2 is not as good as 12.10 because 13.x releases were reorganized to support multiple targets. However we see some improvement from version 13.4 onwards. Most of the tests that do not pass were mainly due to the async clause that we will be discuss in the next section. Figure 8(c) shows the plots for Cray compiler. The bar plots mostly shows no variation, we discuss probable reasons in the next section.

B. Analysis of some bugs identified

In this section, we discuss some of the qualitative and quantitative collection of results about the bugs we found in the OpenACC compilers. A bug can be due to a number of reasons; non-conformance to the specification, compilation, runtime or validation errors.

CAPS: Variable expressions inside gang/worker/vector: OpenACC uses num_gangs, num_workers and vector_length clauses to specify the number of gangs, workers and vector threads that are created to execute on the accelerator. For instance, in an accelerator parallel region, num_gangs(*scalar-integer-expression*) specifies the number of gangs that will execute the region in parallel. The code snippet in figure 9 shows a simplified test case for num_gangs clause allowed on the parallel construct. The basic idea is to specify the number of gang threads to update a shared counter, *gang_num*, and check whether the updated value equals the number of gangs that we set. However, we found that in CAPS compiler versions earlier to 3.1.0, only constant expression inside the num_gangs/num_workers/vector_length were supported, this bug was fixed in the later versions of compiler releases.

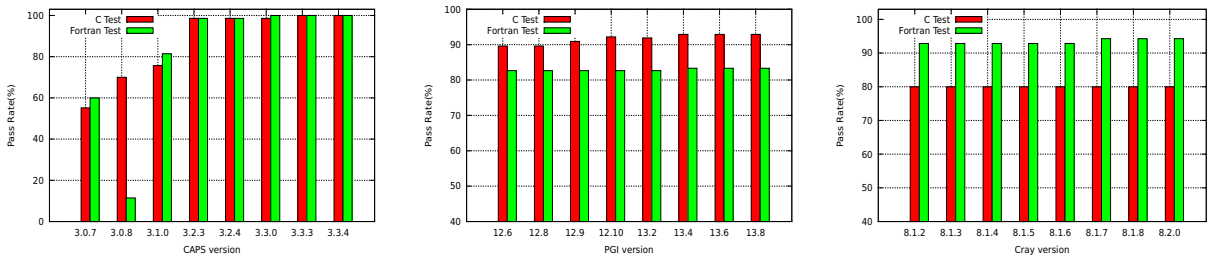
```
int gangs = 8;
int known_gang_num = 8;
int gang_num = 0;

#pragma acc parallel num_gangs(8) reduction(+:gang_num)  /*
     working*/
#pragma acc parallel num_gangs(gangs) reduction(+:gang_num)
      /*not working*/
{
    gang_num++;
}

return (gang_num == known_gang_num);
```

Fig. 9. Test case for num_gangs on parallel construct.

PGI: Asynchronous tests: The acc_async_test() routine is to test the completion of all asynchronous activities. Figure 10 shows a code snippet that tests for acc_async_test(). The basic idea is fairly straightforward: we asynchronously execute a large kernel region on the device and immediately test to check for finished execution. If the asynchronous activities are not completed, 0 is expected to be returned. We then added a wait construct to guarantee that the asynchronous activities have finished execution, and execute the test for acc_async_test() again. At this point a nonzero is expected to be returned. However, we found that PGI 13.x C compilers always returned the value -1 that is what has been set as the initial value. PGI has the same issue with other asynchronous tests such as kernels async, parallel async, wait, acc_async_wait(), acc_async_wait_all() and acc_async_test_all(). Although PGI compiler has the



(a) CAPS (b) PGI (c) CRAY

Fig. 8. Test pass rate for different compilers and their versions

TABLE I
BUGS IDENTIFIED IN DIFFERENT COMPILERS, F: FORTRAN

Compiler	CAPS															
Version	3.0.7		3.0.8		3.1.0		3.2.3		3.2.4		3.3.0		3.3.3		3.3.4	
Language	C	F	C	F	C	F	C	F	C	F	C	F	C	F	C	F
Bugs	36	32	24	70	20	15	1	1	1	1	1	0	0	0	0	0
Compiler	PGI															
Version	12.6		12.8		12.9		12.10		13.2		13.4		13.6		13.8	
Language	C	F	C	F	C	F	C	F	C	F	C	F	C	F	C	F
Bugs	8	14	8	14	7	14	6	14	6	14	5	13	5	13	5	13
Compiler	CRAY															
Version	8.1.2		8.1.3		8.1.4		8.1.5		8.1.6		8.1.7		8.1.8		8.2.0	
Language	C	F	C	F	C	F	C	F	C	F	C	F	C	F	C	F
Bugs	16	6	16	6	16	6	16	6	16	6	16	5	16	5	16	5

same issue with these tests, it can pass all of them if the data clauses are moved out using `data` directive. So it seems that the `async` clause used with `parallel` or `kernel` directive will block the asynchronous activities if they are used together.

```
int is_sync = -1;
#pragma acc kernels copyin(A[0:N], B[0:N]) copy(C[0:N])
    async(tag)
for(i=0; i<N; i++)
  C[i] = A[i] + B[i];

is_sync = acc_async_test(tag); //should be zero, since gpu
    has not done the computation.

#pragma acc wait(tag)

is_sync = acc_async_test(tag); //should be non-zero, since
    there is a "barrier" before
```

Fig. 10. Test case for `acc_async_test`.

Cray: Data copy for scalar variables: In the test for `copy` clause within the `data` construct, we check if the compilers copy scalar variables besides copying array. For instance when we copy a scalar variable from the device to the host, we set a flag to 1 on the device and transfer the data back to the host, but the scalar variable copy does not happen.

Cray compiler behavior: Cray compiler behaves little differently to the other two vendor compilers. Cray performs forward substitution on *flag* and at the compile time determines that flag != HOST and the "if" statement is eliminated. A code snippet is shown in Figure 6. Figure 11 shows a test that ensures that the data in array B will only be copied out and not copied in. We compare the value of array B on the host with the pre-calculated values to ensure this happens. The initial values of array B are randomly generated on the host. On the device, we do not assign any values to the array B (as shown in the code snippet), but we still copy values back to the host as part of the testing process. The copied out values should be different to that of the initial values. The Cray compiler detects the dummy loop as a region where there is no computation going on and deletes the full compute region. Hence the test does not work as expected.

```
 #pragma acc parallel copyout(B[0:N], C[0:N])
 {
    #pragma acc loop
    for(j=0;j<N;j++)
        C[j]=B[j]; //dummy loop
}
for (i=0;i<N;i++)
    sum+=B[i];

if(sum==known_sum)
    error++;
```

Fig. 11. Cray's compilation behavior on `copyout` clause

C. Other Interesting Observations

In this subsection, we also discuss several interesting observations in the OpenACC 1.0 specification we found in the process of developing the testsuite. Most of the issues seem to have been fixed in the OpenACC 2.0.

```
acc_set_device_type(acc_device_not_host);
device_type = acc_get_device_type();
if(device_type != acc_device_not_host)
  fprintf(stderr,"failed on acc_device_not_host\n");

acc_shutdown(acc_device_not_host);
```

Fig. 12. Test case for `acc_set_device_type` as `acc_device_not_host`.

Device type: The OpenACC 1.0 specification defines four types of devices: `acc_device_none`, `acc_device_default`, `acc_device_host` and `acc_device_not_host`, which tells the runtime what type of device to use when executing an accelerator parallel or kernels region. However, during our test for `acc_set_device_type` as `acc_device_not_host`, as is shown in Figure 12, we found that the real device type returned is implementation-defined. For instance, CAPS compiler 3.3.3 considers two additional device types: `acc_device_cuda` and `acc_device_opencl`. PGI version 13.4 considers `acc_device_nvidia`, `acc_device_radeon`, `acc_device_xeonphi`, `acc_device_pgi_opencl` and `acc_device_nvidia_opencl`.

Although the device type name is not specified in the 1.0 specification, 2.0 appendix provides with some recommendations for the device type names for NVIDIA GPU, AMD GPU and Intel Xeon Phi Coprocessor. Although they are not part of the standard, they can make the implementations more easily portable.

Default behavior: The OpenACC 1.0 specification defines a set of data clauses (e.g., `copy`, `copyin`, `copyout`, `present`, `present_or_copy`, etc.). However, if an array referenced in the `parallel` construct does not appear in any data clause, it will be treated as if it appeared in a `present_or_copy` clause. However, the OpenACC 1.0 specification is still missing a `default(`*`list`*`)` clause, which allows users to override the default behavior when data is used in `parallel` or `kernel` region but does not appear in the data clause. In addition, if `present_or_copy` is the default clause used by the compiler, this may introduce unnecessary data movement affecting performance.

In the OpenACC 2.0 specification, a new `default(none)` clause is introduced. It tells the compiler not to implicitly determine a data attribute for any variable, but to require that all variables or arrays used in the compute region do not have any predetermined data attributes.

Procedure calls: Most of the OpenACC compilers are yet to support OpenACC runtime routines and the user-defined routines inside a parallel/kernels region. Lacking such a feature will significantly affect the structure of the OpenACC codes and it is especially inconvenient for large programs.

The OpenACC 2.0 specification added a new `routine` directive which is used to tell the compiler to compile a given procedure inside a `parallel/kernels` region.

Data lifetime: OpenACC uses the `data` construct to allow the programmers to manage the data lifetime on the device. For example, the data copy(a[0:n]) will copy the array a from host to device at the entry of a given code block, and copy it out back to the host at the exit of the given code block. This promotes the structured data lifetime, but not all data lifetime are easily amenable to structured lifetime. Unstructured data lifetime is quite common in large programs with multiple files, where data may be copied into the device in one file while be copied out to the host in another file.

OpenACC 2.0 adds two new directives, `enter data` and `exit data`, which could be used to easily managed the unstructured data lifetime.

Loop nesting: As discussed in Section I, OpenACC uses `gang/worker/vector` clauses to specify different levels of parallelism in the `parallel/kernels` region. Mapping the loop nesting to different levels of parallelism may affect the performance significantly. However, the OpenACC 1.0 specification does not specify the order in which the three clauses can be used; different combinations can lead to different performance results.

OpenACC 2.0 is more strict about loop nesting. The specification says that `gang` loop must be outermost while `vector` loop must be innermost. In addition, a `gang (worker, vector)` loop cannot contain another `gang (worker, vector)` loop unless within a nested `parallel` or `kernels` region. A new added `auto` clause will instruct the compilers to determine the best mapping mechanism.

VI. Discussion

We cover the entire feature set of OpenACC 1.0 specification. Our testsuite consists of over 160 test cases (both C and Fortran). We maintain a tabular column to capture the "pass" or "fail" against each feature implemented by the three compilers. Due to space constraints, we have not included the large table. From the plots in the evaluation section, it is evident that our bug reports helped compiler developers fix the bugs. The most recent releases of the compiler show an increase in the quality (reduced number of bugs identified). We are also confident that the bugs that have been identified are critical requiring immediate attention, since some of these bugs have also been rediscovered and reported by the application developers from national labs.

Software Currently this OpenACC validation suite is under active development by the members of the OpenACC committee. The suite is available for free to any OpenACC member. New academic members can join for a trivial fee. Members wishing to contribute code to the testsuite may do so under a dual license scheme. 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. We welcome contributions.

VII. Production Use

The OpenACC validation suite is being used to validate the functionality of the programming environment of Titan. The suite has been useful to test different OpenACC compilers

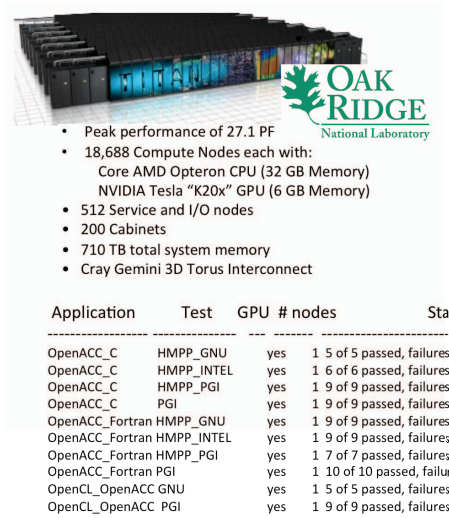


Fig. 13. The OpenACC Validation Suite was used to validate the Titan supercomputer

implementations and to track functionality improvements or degradation over time. The suite runs on random nodes to check functionality requirements of the nodes. It is also used to test different software stacks, for example, to test the translation of OpenACC to CUDA or OpenCL as shown in Figure 13.

VIII. Related Work

To the best of our knowledge, we are the first team to develop a validation suite for OpenACC compilers. The ideas for building the validation suite is adapted from our prior work in [7] and [8].

Some of the other existing commercial test suites such as [9]–[11] are primarily aimed to check for conformance of C and Fortran standards. There are other related efforts that discusses ways and means to detect compiler bugs; not quite validate compilers' conformance to a specification. Csmith [12] is one such approach that performs a randomized test-case generator hunting down compiler bugs using differential testing. The basic idea of randomized differential testing is a black-box approach that automatically generates short test cases that are compiled by various compilers. They run the executable and compare the outputs. 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. We could therefore say that our approach is complementary to that of Csmith's approach.

IX. Conclusion and Future Work

In this paper, we evaluate three commercial OpenACC compilers that are being widely used for porting applications to accelerators. We develop a validation suite that can be used to check OpenACC implementations for conformance to the standard. First, we define and create tests for each individual item in the specification. Second, we have also developed a check for each feature that is designed to test for correct behavior of the implementation. Third, we also facilitate the addition of newer tests, either to cover new features, or test feature combinations, or to test different aspects of an implementation. Fourth, we design cross-tests to increase the confidence in the implementation correctness.

The coverage of tests can be widened by testing several combinations of the features. However as one could imagine, this cannot be a thoroughly complete task since there may be several different permutations and combinations of features co-existing with one another. We have begun to create test cases for 2.0 feature set. We will also at some point soon identify corner cases that are in general quite challenging to be detected manually.

Acknowledgment

We are very grateful to NVIDIA/PGI (Duncan Poole, Yuan Lin, Michael Wolfe, Brent Leback, Pat Brooks and Mathew Colgrove), CAPS (Francois Bodin, Stephane Chauveau, Guillaume Poirier, Yann Mevel and their OpenACC support team), Cray (James Beyer, CR Schult, David Oehmke) and other OpenACC users including Oscar Hernandez, Jean-Charles R, Jeff Poznanovic, Jeff Vetter, Seeyong Lee) for their valuable inputs without which this project could not have been successful. We also want to thank Mike Brim for helping us integrate the OpenACC validation suite to the harness suite of Titan.

References

[1] "PGI Fortran & C Accelerator Compilers and Programming Model," http://www.pgroup.com/lit/pgi_whitepaper_accpre.pdf.
[2] C. Enterprise, "HMPP: A Hybrid Multicore Parallel Programming Platform," http://www.caps-entreprise.com/en/documentation/caps_hmpp_product_brief.pdf.
[3] T. D. Han and T. S. Abdelrahman, "hiCUDA: High-Level GPGPU Programming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 78–90, 2011.
[4] "The OpenACC Application Programming Interface," http://www.openacc-standard.org, Novermber 2011.
[5] "The OpenACC Application Programming Interface, 2.0," http://www.openacc-standard.org/node/297, Novermber 2011.
[6] "Cray enhances coprocessor and accelerator support for openacc 2.0," http://insidehpc.com/2013/10/08/cray-enhances-coprocessor-accelerator-programming-support-openacc-2-0/, October 2013.
[7] M. Müller and P. Neytchev, "An OpenMP Validation Suite," in *Fifth European Workshop on OpenMP, Aachen University, Germany*, 2003.
[8] C. Wang, S. Chandrasekaran, and B. Chapman, "An OpenMP 3.1 Validation Testsuite," *OpenMP in a Heterogeneous World*, pp. 237–249, 2012.
[9] ACE Associated Computer Experts., "SuperTest C/C++ Compiler Test and Validation Suite," http://www.ace.nl/compiler/supertest.html.
[10] Perennial, Inc., "ACVS ANSI/ISO/FIPS-160 C Validation Suite, ver. 4.5, Jan. 1998." http://www.peren.com/pages/acvs_set.htm.
[11] Plum Hall, Inc., "The Plum Hall Validation Suite for C." http://www.plumhall.com/stec.html.
[12] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers," *SIGPLAN Not.*, vol. 46, no. 6, pp. 283–294, Jun. 2011. [Online]. Available: http://doi.acm.org/10.1145/1993316.1993532