An in-depth evaluation of GCC's OpenACC implementation on Cray systems

Verónica G. Vergara Larrea, Wael R. Elwasif, Oscar Hernandez
Computing and Computational Sciences Division
Oak Ridge National Laboratory
Oak Ridge, TN, USA
Email: {vergaravg,elwasifwr,oscar}@ornl.gov

Cesar Philippidis, Randy Allen
Advanced Research Group
Mentor Graphics
Wilsonville, OR, USA
Email: {Cesar_Philippidis,Randy_Allen}@mentor.com

***Abstract*—OpenACC is a directive-based API that extends the C/C++ and Fortran base languages to program accelerators and multicores. Several commercial implementations are available that support OpenACC including PGI, Cray, and PathScale. More recently, GCC started adding support for OpenACC and is expected to fully support the OpenACC 2.0 specification in the upcoming GCC 7 release. However, to our knowledge, the quality and performance of GCC's OpenACC implementation have not been studied in detail.**

In this paper, we will perform an in-depth evaluation of GCC's OpenACC implementation on Titan, ORNL's Cray XK7 supercomputer, and compare it to other commercially available compiler implementations. We first start by providing a description of the OpenACC implementation design in GCC, its runtime, as well as provide an overview of the current state of OpenACC supported features as described in GCC 6.3. Then, we we will evaluate the quality and performance of the GCC 6.x implementation by using the OpenACC Verification and Validation suite [1] to test the accuracy and correctness of the implementation, the EPCC OpenACC benchmark suite [2] to measure performance, and the SPEC ACCEL benchmark [3] OpenACC suite to exercise the implementation. We believe that the results presented in this study will be useful for the larger community interested in using and evaluating new OpenACC implementations.

***Keywords*-OpenACC; compiler evaluation;**

I. Introduction

OpenACC is a relatively new directive-based specification to program accelerators and for that reason, its compiler implementations are still maturing. Several commercial implementations support the OpenACC standard including PGI, Cray, and PathScale. More recently, GCC started adding support for OpenACC in an effort to fully support the OpenACC 2.0 specification. The latest release of GCC is GCC 6.3 and it already includes partial support for OpenACC, with full-support for OpenACC expected to be available in the upcoming GCC 7 release. GCC 8 will include support for OpenACC 2.5. However, to our knowledge, the quality and performance of the GCC OpenACC implementation have not been explored in detail.

Several benchmark suites have been developed to evaluate OpenACC using different compilers on different architectures such as: the SPEC ACCEL benchmark suite [3], the EPCC OpenACC benchmark suite [2], the University of Houston's Verification and Validation OpenACC suite [1], the KernelGen performance test suite for OpenACC [4] compilers, among others. These benchmarks are developed to understand the level of specification support, its quality (e.g. overheads), and to measure its performance against other implementations, which in turn allows researchers to compare compilers on different architectures.

When looking at the two main architectural trends, attached accelerators and many-core processors, it is clear that high-level programming models, such as OpenACC 2.5 and OpenMP 4.5, should be able to support both trends well for performance portability. As of today, implementations like PGI support both *offloading* to an accelerator, and *self-offloading* where OpenACC parallel regions and/or kernels are launched on the host rather than on an accelerator. It is not understood how well these two modes are supported by the compilers and whether they can produce performant code. Because of this, and keeping performance portability in mind, this study briefly discusses experiments conducted on ORNL's Cray XC40 Intel Knights Landing system using supported compilers for *self-offloading*.

In this paper, we first start by providing a description of the OpenACC implementation design in GCC, its runtime, as well as provide an overview of the current state of OpenACC supported features as described in GCC 6.3 and those available in the *gomp-4_0-branch* branch maintained by Mentor Graphics, a Siemens company. In order to evaluate the quality and performance of the GCC 6.3 implementation, we will first use the OpenACC Verification and Validation suite [1] to test the accuracy and correctness of the implementation. We will also use the EPCC OpenACC benchmark suite to measure the performance of OpenACC functions (e.g. the time it takes to copy data to or from the device, overheads of using the *private* and *firstprivate* clauses), and the performance of compute intensive kernels. Furthermore, we will use the SPEC ACCEL OpenACC benchmark suite [3] to exercise the GCC implementation using application-based benchmarks and compare these against published SPEC ACCEL results. The experiments for this evaluation will be conducted on Titan, ORNL's Cray XK7 flagship supercomputer. We believe that the OpenACC results presented in this study will be useful to the community in general given the fact that GCC is widely available on multiple platforms and Linux distributions. At the end of the paper we will have a discussion on how having an OpenACC implementation in GCC makes it easier to support OpenMP 4.5 (offload) as both can share the same runtimes, which in turn may help the interoperability of both specifications or leverage one from the other.

A. Accelerator Programming Model using Directives

OpenACC implements a programming model that has a host with one or more attached accelerators. The user application begins execution on the host and accelerated regions are offloaded to the accelerator device under control of the host (offload execution). The host is typically a general purpose processor (CPU) that can offload code to other devices or, in some implementations, that can offload code to itself. The device executes OpenACC parallel and kernels regions, which typically contain work-sharing regions of the code executed as accelerated kernels. The host coordinates the execution of the accelerated kernels by allocating memory on the accelerator device, initiating data transfer, sending the code to the accelerator, passing arguments to the accelerated region, queuing the device code, waiting for completion, transferring results back to the host, and deallocating memory. The host can queue one or more sequences of accelerated kernels to be executed on the device, either sequentially, one after the other, or asynchronously.

B. Accelerator Memory Model

OpenACC has a copy-in and copy-out memory model that is synchronized with the host memory. All data movement between host and device memory is performed by the host through runtime library calls that move data between memories (e.g. host and accelerator memories). However, from the user's perspective, most of the data mapping is handled either implicitly by the compiler or explicitly via data clauses. OpenACC provides data regions to scope the data that is going to be copied-in, updated, and copied-out to or from the accelerator.

C. Accelerator Execution Model

OpenACC expresses the levels of parallelism via *gang*, *worker*, and *vector* threads. Each gang contains one or more workers. Each worker can contain one or more vector threads. Users can specify the size of gangs, workers, and vectors or allow the compiler to select them automatically. When an *acc parallel* or *acc kernels* region is launched, a group of gangs, workers, and vector threads are created on the device. All start executing in gang-redundant, worker-single, and vector-single mode until work-sharing constructs are encountered (e.g. an OpenACC loop) indicating that work should be distributed among the different threads. In OpenACC, the *acc parallel* directive begins the execution of a single accelerator kernel on the device. OpenACC also provides an *acc kernels* directive. Like *acc parallel*, *acc kernels* denotes a region of code to be offloaded to an accelerator. However, whereas *acc parallel* requires the user to explicitly mark independent loops with *acc loop* directives, some compilers can often automatically detect and parallelize independent loops inside *acc kernels* regions. The *acc parallel* directive gives the user a finer grain of parallelism, which is necessary to achieve good performance on a given architecture when the compiler cannot automatically parallelize a given region. Both OpenACC *parallel* or *kernels* regions can be executed as asynchronous on the device(s) using the *async* clauses and synchronized with the *wait* directive. OpenACC also allows nested parallelism where an OpenACC *parallel* region spawns another *parallel* region on a device to achieve more levels of parallelism. However, support for nested parallelism is not currently available in the implementations used in this work.

II. Overview of OpenACC Support in GCC

This section presents a high level overview of the support for OpenACC in GCC. GCC is a widely used, open source, retargetable compiler that supports the C, C++, Fortran, Go, and Ada programming languages. As of GCC 6, it supports a large subset of CilkPlus, OpenACC 2.0a, and OpenMP 4.5 parallel programming methodologies.

Part of the benefit of being an open source project is that a lot of functionality is shared between various components. In fact, GCC's support for OpenACC was built on top of its existing support for OpenMP. Note however that, due to the vastly different organizations of GPU and CPU hardware, including SIMD vs. SIMT type of parallelism differences, limited stack size and local memory; extensive modifications were required to implement OpenACC efficiently on GPUs. While that functionality is not currently enabled in OpenMP 4.5 target offloading, there are no technical barriers preventing anyone from leveraging from the OpenACC work to do so in the future.

Mentor Graphics is currently maintaining the OpenACC port in GCC and Oak Ridge National Laboratory is providing feedback on Mentor Graphics implementations [5]. The first GCC release that included support for OpenACC 2.0a was version 5. Support for OpenACC in that version was highly experimental and restricted all parallelism to vectors.

In GCC 6, Mentor Graphics contributed a more complete execution model that supports gang, worker, and vector level parallelism inside OpenACC parallel regions, in addition to preliminary support for OpenACC routines. For GCC 7, Mentor Graphics has started focusing on performance, in addition to refining the support for OpenACC 2.0a routines and the *declare* directive. Looking ahead, the GCC 8 release will include support for OpenACC 2.5. Mentor Graphics is also focusing on making the OpenACC experience more user friendly by extending GCC with enhanced compiler diagnostics to inform the user of any potential parallelism enhancements, which is something that GCC currently lacks.

As of the time of this writing, users interested in evaluating GCC's support for OpenACC are highly recommended to obtain a release based on the *gomp-4_0-branch* branch. That branch is currently based on GCC 6.3 and it contains additional OpenACC functionality and enhanced performance on NVIDIA GPUs. Details on how to obtain a GCC toolchain using that branch can be found in [6].

A. Known limitations in GCC

While the current release of GCC does support OpenACC 2.0a, there are still several limitations. First, GCC only supports OpenACC offloading on NVIDIA GPUs. Any other target (e.g. multicore host) falls back to executing on a single host CPU thread.

The following is a list of the current known limitations:

- Nested parallelism is not supported, i.e. parallel constructs cannot nest inside other parallel constructs.
- While GCC is able to automatically partition independent *acc loop*s across gang, worker, and vector dimensions, at times, this optimization can be too aggressive. When that happens, executables generated by GCC may fail at runtime with errors involving insufficient hardware resources.
- GCC supports dynamic arrays in C and C++ inside OpenACC data clauses with the following limitations.
  - The pointer-to-arrays case is not supported yet, e.g. int (*a)[100].
  - Host fallback does not work yet, i.e. ACC_DEVICE_TYPE=host will generate a segmentation fault.
- All OpenACC loop private clauses allocate storage for variables in local (i.e. thread-private) storage. They will utilize shared memory storage in a future release.
- GCC does not support the *device_type* clause.
- *vector_length* is fixed to 32 for NVIDIA targets, and 1 for everything else.
- The *bind* clause has not been implemented yet.
- The *private* and *firstprivate* clauses do not support subarrays.

Unlike more mature OpenACC compilers, GCC is frequently unable to detect parallelism inside OpenACC kernels regions. The parallelism obtained from adding the *acc kernels* directive is strictly implementation dependent, and therefore when the compiler is able to detect parallel loops, better performance will be achieved. The GCC implementation does not currently support auto-parallelization, and because of that kernel regions will result in slower performance. With GCC, better performance will be obtained by replacing kernel regions with parallel regions [6]. If the compiler fails to detect any parallelism inside a kernels region, it falls back to executing that region on a single host thread. Furthermore, because GCC does not support auto-parallelization, lower performance may be obtained when using *acc parallel* loops without more specific directives that indicate the level of parallelism (e.g. *gang*, *worker*, and *vector*) of a region [6].

III. OpenACC Implementation in GCC

GPUs are organized vastly different from conventional out-of-order CPUs. For starters, they often have a small stack size which limits recursions and function call depth. GPUs also have limited interprocess communication capabilities which impacts thread synchronization, and have a minuscule amount of local memory when compared to the CPU. Perhaps, most significantly, they form a heterogeneous parallel environment when combined with the host processor, which necessitates managing data in discrete memory address spaces. Furthermore, as is the case of OpenCL and CUDA, end users have to use runtime libraries to explicitly manage the data mappings of variables. In addition, while these programming environments are largely compatible with C and C++, they frequently require language extensions to invoke functions that are to be executed on the accelerator.

OpenACC was designed early on to map the existing C, C++, and Fortran programming language semantics to GPU computing through the use of compiler directives. Consequently, this enables a well-formed OpenACC program to generate identical results when it was built with and without OpenACC enabled by the compiler. One key advantage is that OpenACC programs can still be built with compilers that do not support those directives, which improves code portability.

A. Implementing OpenACC to target GPUs

As described in previous sections, OpenACC provides three levels of parallelism: gang, worker, and vector. Each level may operate in two modes; gangs can operate in redundant or partitioned mode, whereas workers and vectors can run in single or partitioned mode. Partitioned mode maps each gang, worker, or vector unit to a single loop iteration. Worker and vector single modes only use one of the available worker or vector units, respectively, when they execute code, i.e. the worker and vector units as a whole are treated as an entity containing a single processor. In gang redundant mode, all the gangs execute the code without any synchronization in a redundant fashion.

The justification for gang redundant mode stems from the fact that some accelerators, like GPUs, often have limited interprocess communication facilities. GPU hardware is designed to execute thousands of threads concurrently. In order to scale the hardware from handheld devices, such as cellphones and tablets, to powerful supercomputers, hardware vendors partitioned their GPUs into multiprocessor clusters (MPC). The more powerful GPUs have more MPC units, whereas handheld devices have fewer. In NVIDIA hardware, an MPC is called a Streaming Multiprocessor (SM). Each MPC consists of an array of tightly connected processors. While each processor has its own instruction pointer register, for optimal performance, it is better if all processors in an MPC execute a common instruction. When the processors within an MPC execute different instructions, the MPC is said to be divergent. Finally, each individual MPC processor is typically capable of executing multiple threads, hence the name *single instruction multiple threads (SIMT)*. Because there is no requirement for each MPC to be convergent with one another, GPU hardware often omit intra-MPC synchronization mechanisms. In fact, sometimes different MPCs execute different kernels to more effectively utilize the GPU hardware. CUDA does not provide a function to synchronize CUDA blocks, where each thread inside a block gets assigned to a single MPC. Note that an MPC may execute multiple blocks.

When developing parallel code, the developer must be aware of how the compiler maps the available OpenACC parallelism to the underlying hardware. GCC uses the following:

- Gangs are mapped to CUDA thread blocks.
- Workers are mapped to CUDA warps.
- Vectors are mapped to CUDA threads.

A CUDA block consists of a collection of CUDA warps. Each CUDA warp contains a fixed amount of CUDA threads. Current NVIDIA hardware has a warp size of 32 threads. As a consequence of this mapping, GCC fixes *vector_length* to the size of the warp, i.e. 32 threads. Each CUDA block has its own resources which are shared among the warps and threads within it. One such resource is shared memory, which GCC uses for gang-local variables, worker reductions, and some internal state management.

While GCC can generate code for NVIDIA accelerators with little to no modifications inside parallel regions, the user can often greatly assist the compiler explicitly by adding OpenACC loop directives as the following examples illustrate.

B. Porting Matrix Multiplication to OpenACC

Consider the matrix multiplication example shown in Listing 1, where *a*, *b*, and *c* are one dimensional arrays, *n* is the matrix dimension and is set to 2,048, and *at* is the macro defined as shown in Listing 2.

```
for (i = 0; i < n; i++)
{
  for (j = 0; j < n; j++)
  {
    int t = 0;

    for (k = 0; k < n; k++)
      t += at(i, k, a) * at(k, j, b);

    at(i, j, c) = t;
  }
}
```

Listing 1: Matrix Multiplication

```
  #define at(y, x, mat) (mat[y*n + x])
```

Listing 2: *at* macro definition

On Titan using the GCC-gomp4 compiler, this loop takes approximately 180.11 seconds to execute using a single host thread. Note how this is significantly slower than the host CPU.

OpenACC provides two ways to accelerate this code, using parallel or kernels regions. Throughout this example, we will only utilize OpenACC parallel regions.

1) Adding parallel regions: Simply adding *acc parallel* will direct GCC to generate code for the host and accelerator. The resulting code is shown in Listing 3.

```
#pragma acc parallel
for (i = 0; i < n; i++)
{
  for (j = 0; j < n; j++)
  {
    int t = 0;

    for (k = 0; k < n; k++)
      t += at(i, k, a) * at(k, j, b);

    at(i, j, c) = t;
  }
}
```

Listing 3: Matrix multiplication using the *acc parallel* construct

Notice that this loop nest does not contain any explicit *acc loops*. By default, GCC will execute this parallel region with 1 gang, and 32 vectors operating in vector-single mode, i.e. this entire parallel region will be executed using a single CUDA thread. Note that if the user explicitly sets *num_gangs* to a number greater than one, the resulting binary will yield unpredictable results because the entire parallel region will execute in gang redundant mode.

On Titan using GCC-gomp4, this version of the code takes approximately 3210.56 seconds to execute.

2) Adding parallel loops: Explicitly marking independent loops inside OpenACC parallel regions with *acc loop* is required to activate any partitioned mode inside a parallel region using GCC. The *acc loop* construct enables sharing work among gangs, workers, and vector threads. GCC does not attempt to automatically parallelize parallel regions. It does, however, attempt to assign gang, worker, and vector parallelism to independent loops. Furthermore, it also sets minimum default values for *num_gangs*, *num_workers*, and *vector_length*. If the parallel region only contains one independent loop without any explicit *gang*, *worker*, or *vector* directive, GCC will automatically assign gang and vector parallelism to the region. On the other hand, if a parallel region contains multiple loops, the innermost loop will be assigned vector level parallelism, and the outermost loops will be assigned gang and worker level parallelism [7]. GCC will report the assigned level parallelism using the *-fopt-info-note-omp* flag.

Listing 4 shows the matrix multiplication example with the addition of the *acc loop* construct immediately following the *acc parallel* construct. Here, GCC assigned gang and vector level parallelism to the *acc loop* automatically.

```
#pragma acc parallel
#pragma acc loop
for (i = 0; i < n; i++)
{
  for (j = 0; j < n; j++)
  {
    int t = 0;

    for (k = 0; k < n; k++)
      t += at(i, k, a) * at(k, j, b);

    at(i, j, c) = t;
  }
}
```

Listing 4: Matrix multiplication using *acc parallel* and *acc loop* constructs

On Titan using GCC-gomp4, this change reduces the execution time to 4.25 seconds.

3) Adding multiple parallel loops and specialized constructs: In Listing 4, only one parallel loop was added, so effectively, GCC only utilized one level of parallelism. In Listing 5, two more *acc loop*s are added, and one of them contains a reduction.

```
#pragma acc parallel present (a[0:n*n], \
b[0:n*n], c[0:n*n])
#pragma acc loop
for (i = 0; i < n; i++)
{
  #pragma acc loop
  for (j = 0; j < n; j++)
  {
    int t = 0;

    #pragma acc loop reduction (+:t)
    for (k = 0; k < n; k++)
      t += at(i, k, a) * at(k, j, b);

    at(i, j, c) = t;
  }
}
```

Listing 5: Matrix multiplication using *acc parallel*, *acc loop*s, and *vector reduction*.

Here, the GCC compiler applied gang partitioning to the outermost loop, worker partitioning to the middle loop, and vector partitioning to the innermost loop containing the reduction. On Titan using GCC-gomp4, this version of the matrix multiplication code takes only 1.83 seconds. Table I summarizes all the results for the different versions of the code.

Code version	Time (s)
Sequential	180.11
Parallel	3210.56
Parallel Loop	4.25
Parallel Loops/Reduction	1.83

Table I: Matrix Multiplication timings on Titan using the GCC-gomp4 compiler.

In order to compare the behavior of the compilers used in this study, the same procedure was repeated with GCC 6.3 upstream and PGI 17.1 on Titan. While a comparison using a more modern GPU (e.g. P100) would be more appropriate, ORNL does not currently have a P100-based Cray system. The timings obtained with each tested compiler are shown in Figure 1. The speedups obtained at each step of the porting process measured on Titan relative to the sequential timing are shown in Figure 2. The results show that on all cases except for experiments run on Titan using PGI 17.1, adding an *acc parallel* region without *acc loop* directives results in slower code. This is due to the fact that GCC does not autoparallelize the code. The highest speedup is obtained with GCC-gomp4 after the code is fully ported to OpenACC. On Titan, GCC-gomp4 was also able to achieve the fastest version of the code. It is also worth noting that adding an *acc parallel* section resulted in code approximately 20x slower than the serial version. This is due to the fact that in GCC, without the *acc loop* construct, the entire block is executed by a single CUDA thread.

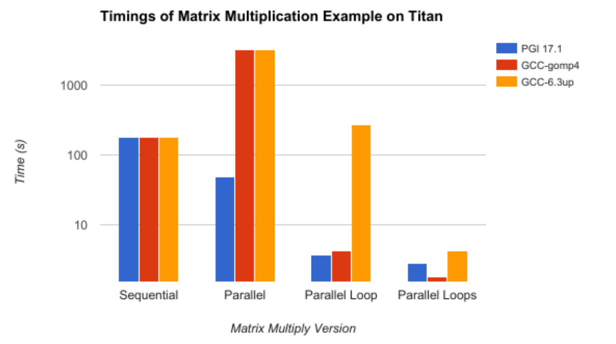

Figure 1: Timings of matrix multiplication code versions on Titan using GCC gomp4, GCC 6.3 upstream, and PGI 17.1.

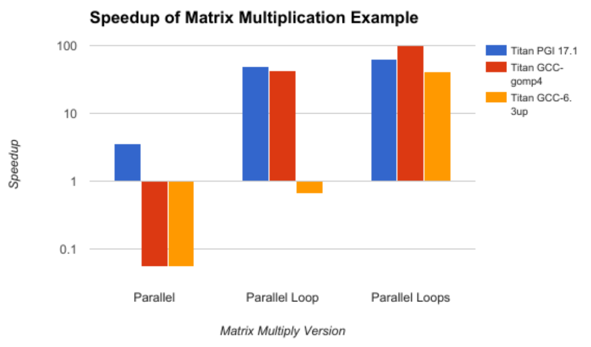

Figure 2: Speedup of matrix multiplication code versions on Titan using GCC gomp4, GCC 6.3 upstream, and PGI 17.1.

IV. Experiments

Several benchmark suites were used in this study in order to evaluate GCC's OpenACC implementation or GCC-gomp4. Ideally, the evaluation should be able to measure: how well (or poorly) does the compiler follow the most current OpenACC specification, what is the performance of the compiler constructs when compared to mature compilers, and what is the impact in performance of kernels and applications when the compiler is used. To meet the first criterion, University of Houston's OpenACC Verification & Validation suite was used. The EPCC OpenACC benchmark suite was used to measure and compare overheads for common OpenACC constructs. Finally, the SPEC ACCEL OpenACC benchmark suite and the KernelGen performance suite were used to look at the performance of the compiler. This section provides a description of each suite used and explains the choices of tests made.

For all the experiments, Titan [8], ORNL's flagship Cray XK7 supercomputer was used. Compute nodes on Titan contain a 16-core AMD Interlagos processor with a peak flop rate of 140.2 GF and a peak memory bandwidth of 51.2 GB/s, and an NVIDIA Kepler K20X GPU with a peak single/double precision flop rate of 3.935/1.311 TF and a peak memory bandwidth of 250 GB/s. For experiments on Titan three compilers were used: Mentor Graphics' GCC gomp4_0-branch (GCC-gomp4), GCC 6.3.0 upstream release (GCC 6.3-up), and PGI 17.1.

Experiments were also ran on Percival, ORNL's XC40 Intel Knights Landing system using the PGI 17.1 compiler and the *-ta=multicore* flag. However, given that GCC's OpenACC implementation does not currently support multicore architectures, we have not included Percival results in this study.

A. OpenACC Validation & Verification Suite

The OpenACC validation test suite [1] is a suite of micro-tests that aims at validating OpenACC implementations of various OpenACC directives against the specification. The suite consists of test cases for all directives supported in version 1.0 of the OpenACC specification and many of the directives included in version 2.0 of the specification. This suite is currently being updated to reflect the changes and additions to the OpenACC specification that are part of version 2.5, however this updated version is expected to be released later this year and so it was not used in this paper.

B. EPCC OpenACC Benchmark Suite

The EPCC OpenACC Benchmark Suite introduced in 2013 was designed to measure and compare the performance of OpenACC compilers running on different architectures [2].

The suite contains three different levels of kernels. *Level 0* tests measure overheads introduced when using certain OpenACC constructs. *Level 1* tests measure the performance of 13 distinct computationally intensive linear algebra kernels. Lastly, the suite also contains 3 kernels that are representative of real-world applications that form the *Level 2* tests. All the kernels and tests in the suite are written in C. The suite has been used in the past to evaluate performance of OpenACC implementations [9].

Because the suite has not been updated recently, there were a few tests that did not work correctly with the compilers used in this evaluation. All *Level 2* tests resulted in runtime errors, even with the PGI compiler which is the more mature implementation among the three tested. Furthermore, because of known limitations of the GCC-gomp4 compiler, seven *Level 1* tests were excluded. The *Parallel_private* and *Parallel_1stprivate* tests were not used because GCC-gomp4 does not yet support subarrays within the *private* or *firstprivate* clauses. In addition, all tests that use the *acc kernels* directive were excluded.

C. SPEC ACCEL Benchmark Suite

The SPEC ACCEL benchmark was developed by the Standard Performance Evaluation Corporation (SPEC) High Performance Group (HPG) to measure the performance of compute intensive parallel applications on accelerators. The latest version of the SPEC ACCEL benchmark is v1.1 and was released in September 2015 and contains two separate benchmark sets. The first set contains 19 application kernels that use OpenCL, and the second contains 15 application kernels that use OpenACC. In this work, we will only look at the latter.

The SPEC ACCEL OpenACC suite is comprised of seven C, six Fortran, and two combined C and Fortran application kernels. The OpenACC suite includes tests from the NAS Parallel Benchmarks [10], the SPEC OMP2012 OpenMP benchmark suite [11], and other representative applications from the high performance computing (HPC) community.

Each benchmark comes with three distinct data sets: test, train, and reference (ref). The reference data set is used to compare the performance of benchmarks across architectures.

Due to the known limitations of GCC's OpenACC implementation described in Section II, only three benchmarks part of the SPEC OpenACC suite were used: 304.olbm, 314.omriq, and 360.ilbdc. The remaining 12 application kernels make use of features not currently supported by the GCC gomp4 compiler, namely *acc kernels*.

1) 304.olbm: The 304.olbm benchmark is written in C and it implements the Lattice Boltzmann Method (LBM). The benchmark was derived from the SPEC CPU2006 [11] 470.lmb and Parboil 2.5 benchmarks [12].

2) 314.omriq: The 314.omriq benchmark is written in C. It was derived from the MRI-Q benchmark part of the Parboil benchmark suite [12], [13]. The benchmark performs an MRI reconstruction.

3) 360.ilbdc: The 360.ilbdc benchmark is written in Fortran and was ported to OpenACC from the SPEC OMP2012 [11] 360.ilbdc benchmark. The kernel implements the collision-propagation routine of an advanced 3D LBM solver.

D. KernelGen OpenACC Performance Suite

The KernelGen performance test suite [4] is a suite of OpenACC codes that was developed as part of the KernelGen project [14] that aimed at developing a prototype of auto-parallelizing Fortran/C compiler for NVIDIA GPUs, targeting numerical modeling code. The KernelGen project itself appears to be dormant, however the performance test suite continues to be a useful tool for evaluating the ability of compilers to exploit “easy” parallelism in numerical code.

The suite [4] consists of several typical single precision numerical algorithms on 2D or 3D regular grids. Each algorithm is performed in 2-3 parallel spatial loops enclosed into a non-parallel time iterations loop. Tests are partially adopted from [15]. We used a modified version of the suite that can be found at [16]. The test suite consists of the following tests along with the programming language used for the implementation

- `divergence`: 3-D divergence operator code (C).
- `gameoflife` Implementation of Conway's game of life (C).
- `gaussblur` 25-point Gaussian blur approximation (C).
- `gradiaent` 3-D gradient operator code (C).
- `jacobi` Jacobi method iterations (Fortran)
- `lapgsrb` 25-point approximation of Laplace operator (C)
- `matmul` Matrix-matrix multiplication (Fortran)
- `sincos` 3D implementation for $z = sin(x) + cos(x)$ (Fortran)
- `tricubic` Tricubic interpolation (C)
- `tricubic2` Tricubic interpolation with alt values grouping (C)
- `uxx1` Variant of scheme for second derivative (C)
- `vecadd` Array sum (C)
- `wave13pt` 13-point 2-time levels explicit scheme for wave equation (C).

The tests were modified slightly to update OpenACC syntax that is not supported in the 2.5 version of the specification (e.g. the use of *gang(65535)* clause) and to address any compiler errors due to incomplete specification of variables in acc parallel regions. Furthermore, the use of *acc kernels* was changed into *acc parallel* to accommodate the existing limitations in the GCC compiler as outlined in Section II.

V. Results

A. OpenACC V&V Results

Table II shows the results obtained from running the OpenACC Verification and Validation suite v1.0 using GCC-gomp4, PGI 17.1, and CCE 8.5.5.

	Passed	**Failed**	**CE**	**RE**	Total
GCC-gomp4	163	17	56	57	293
PGI 17.1	204	19	20	51	294
CCE 8.5.5	158	27	38	72	295

Table II: Results from OpenACC Verification & Validation suite.

While a complete analysis of all failures is beyond the scope of this paper, we note that GCC-gomp4 performs roughly on par with the Cray compiler in terms of total successful tests. Many of the errors reported by GCC-gomp4 reflect a sometimes stricter interpretation of the OpenACC specification. For example 18 compile time errors originate from code such as that shown in Listing 6.

```
int A[N],B[N],C[N], D[N];
#pragma acc declare \
pcopy(A[0:N], B[0:N], C[0:N], D[0:N])
```

Listing 6: Array Sections in *acc declare*.

Where GCC-gomp4 flags the use of array sections *A[0:N]*, *B[0:N]* and *D[0:N]* in the *acc declare* directive. The PGI compiler does not flag the same code, indicating a possible different reading of the specification. The inclusion of GCC into the mix of compilers that support OpenACC, and the development of an updated version of the OpenACC validation suite would help to clarify ambiguities in the specification and fill any gaps that may exist in their testing.

Another source of GCC-gomp4 runtime errors stems from the compiler's sometimes aggressive optimization, which may result in over allocation of resources on the GPU. This results in error messages such as *libgomp: The Nvidia accelerator has insufficient resources to launch 'routine_with_name_$_omp_fn$0'; recompile the program with 'num_workers = 24' on that offloaded region or '-fopenacc-dim=-:24'*. This error is responsible for 4 runtime errors in Table II, and better resource management during optimization is one of the current development thrust areas of the compiler.

B. EPCC OpenACC Benchmark Results

Figure 3 shows the results obtained from executing *Level 0* data movement centric tests on Titan using the default data size transfer of 2MB. When copying contiguous data, the PGI compiler spends the least amount of time in the data region for the three data sizes tested. Both GCC-gomp4 and GCC 6.3 upstream, take a similar amount of time to transfer the data from the host to the device, and vice-versa. For the non-contiguous cases, however, GCC-gomp4 results in the fastest time when copying data from the GPU to the host; whereas PGI results in the fastest time when copying data from the host to the GPU.

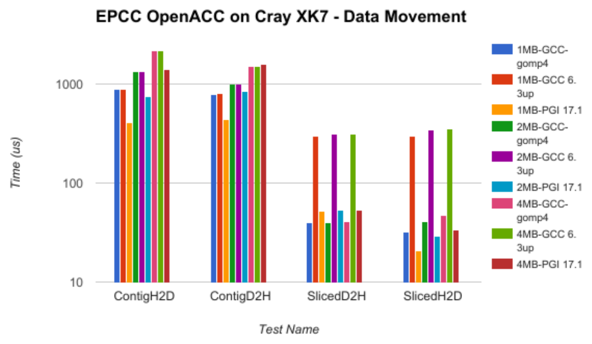

Figure 3: EPCC OpenACC benchmark: Data movement overheads from host to device, and device to host on Cray XK7 system. Lower values represent better performance.

For the remaining *Level 0* tests, the benchmark suite reports the time difference between two slightly different constructs. For example, in the case of the *Parallel_Reduction* test shown in Listing 7. The test reports the difference between computing a *acc parallel loop* and computing a *acc parallel loop reduction* construct. A negative value indicates that the first construct took longer than the first.

```
#pragma acc data copyin(a[0:n])
{
  t1_start = omp_get_wtime();
#pragma acc parallel loop reduction(+:z)
  for (i=0;i<n;i++){
    z += a[i];
  }
  t1_end = omp_get_wtime();

  t2_start = omp_get_wtime();
#pragma acc parallel loop
  for (i=0;i<n;i++){
    z += a[i];
  }
  t2_end = omp_get_wtime();
}
...
  return( (t2_end-t2_start) - (t1_end-t1_start) );
}
```

Listing 7: Code snippet from *Parallel_Reduction* test

Results obtained from executing the remaining *Level 0* tests are shown in Figure 4. In the case of *Parallel_If*, GCC-gomp4 incurs in a smaller overhead than the other two compilers. The same behavior is repeated for all data sizes tested because additional time is spent in the loop as the loop increases in size. For the *Parallel_Combined* test, the GCC-gomp4 compiler shows the least overhead, closely followed by the PGI compiler. The GCC upstream compiler shows an order of magnitude higher overhead for this test. The results from the *Update_Host* test show that GCC gomp4 has greatly improved the performance of the operation in comparison with GCC upstream. The PGI compiler shows the smallest amount of overhead when using the *acc update host*. For the *Parallel_Invocation* test, the overhead increases proportionally with the data size for both the GCC-gomp4 and the PGI compilers. The PGI compiler shows the smallest overhead, and the GCC upstream compiler the largest. The *Parallel_Reduction* test shows that all compilers incur in additional overhead when using a reduction operation. The GCC-gomp4 compiler incurs the highest overhead among the three compilers, in fact, it is consistently an order of magnitude larger than the other two compilers. Among the three, PGI incurs in the smallest overhead. The overhead is explained due to the fact that reductions require and additional step to collect the results from all the private versions of the reduction variable. In GCC-gomp4, the overhead will vary depending on the type of reduction used. *gang* and *worker* reductions utilize CUDA atomics operations, whereas *vector* reductions use CUDA warp shuffle instructions which are faster.

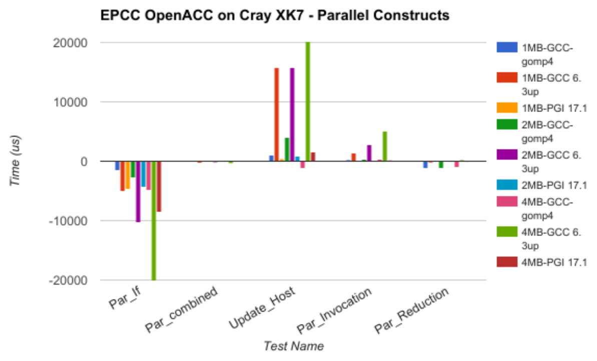

Figure 4: EPCC OpenACC benchmark: Parallel constructs overheads on Cray XK7 system. The figure shows the difference in time between two regions or loops.

Results obtained from executing *Level 1* tests on Titan using the GCC gomp4, GCC 6.3 upstream, and PGI 17.1

compilers are shown in Figure 5. For all compute intensive linear algebra kernels, the PGI 17.1 compiler performed the best, followed by GCC-gomp4.

The results from *Level 1* tests also demonstrate the improvements available in the GCC-gomp4 branch of the compiler when compared to the latest release.

C. SPEC ACCEL OpenACC Suite Results

Three benchmarks part of the SPEC ACCEL OpenACC suite were used to measure the performance of GCC-gomp4 against that of the GCC 6.3-up and PGI 17.1 compilers. Figure 6 shows the timings obtained from running benchmarks 304.olbm, 314.omriq, and 360.ilbdc using the reference data sets. Please note that these timings are measured estimates.

The estimates measured show that the performance of GCC-gomp4 against the PGI 17.1 compiler for the SPEC ACCEL OpenACC benchmarks on Titan. Table III summarizes the impact in performance observed when comparing the two compilers.

Benchmark	**Perf. Diff. Ref**
304.olbm	11.48%
314.omriq	-100.00%
360.ilbdc	-51.21%

Table III: Performance difference between the PGI 17.1 and the GCC-gomp4 compiler. A positive number means GCC-gomp4 performs better than PGI 17.1.

D. KernelGen OpenACC Suite Results

The performance results for the KernelGen performance suite are shown in Figure 7. In addition to the software environment described in Section IV, the suite was executed using with the "standard" optimization flag (`-O3`) enabled and disabled for PGI 17.1 compiler and GCC-gomp4. The mainline GCC 6.3 was executed using with the optimization flag enabled. In these runs, the mainline GCC 6.3 failed to successfully run the `lapgsrb` and `uxx1` tests while the tests were successfully completed using the GCC-gomp4 version. Furthermore, the plot shows the performance improvements that have been implemented in the GCC-gomp4 branch, and which will be reflected in the upcoming GCC 7.0 release. One observation is that enabling optimization makes a bigger difference with the GCC-gomp4 compiler than with PGI. This is because PGI enables `-O2` by default, whereas GCC uses `-O0`.

VI. Discussion

The results presented here clearly show the performance improvements that have been added to GCC in the *gomp-4_0-branch*. The GCC-gomp4 compiler outperformed PGI when running the 304.olbm (ref data sets) SPEC ACCEL OpenACC test on the NVIDIA K20X GPUs. GCC-gomp4 was also able to achieve the highest speedup for the simple matrix multiplication example presented in Section III. There are several features that are not currently available in GCC-gomp4 described in Section V. As the implementation continues to mature and more features are added, a full feature evaluation could be conducted.

One particular advantage of the GCC OpenACC implementation is that it relies on the gomp4 runtime which is shared with the OpenMP 4.5 implementation. In the paper [17] we describe the differences between OpenMP 4.5 and OpenACC 2.5. The majority of directives expressed in OpenACC 2.5 can be lowered to OpenMP 4.5 as an intermediate translation, which then can be lowered to a common runtime. The Cray compiler, for example, uses the same OpenMP and OpenACC runtimes. This is beneficial because any bug fixes or improvements that may be done in the runtime to support OpenACC, will also have a positive impact in the OpenMP 4.5 implementation.

A. OpenACC benchmarks

As it was highlighted in the description of tests in Section IV, several benchmarks including the EPCC OpenACC benchmark suite, and OpenACC V&V suite have not been updated recently. While there is a clear need to be able to validate and verify an implementation against the specification, as the OpenACC V&V suite attempts to do, if the benchmark is not kept current with the latest specification, the tests become less valuable. Furthermore, as described in Section V, as the specification changes, some tests will inevitably become not compliant with the current specification.

Benchmark suites must also be regularly tested. As evidenced by the results from the EPCC OpenACC benchmark suite, all *Level 2* tests produced CUDA errors. This could point to a bug in the implementation tested, but it could also be a bug with the test. Because SPEC ACCEL OpenACC tests can serve as substitutes of real-world kernels, excluding EPCC OpenACC *Level 2* tests did not greatly impact this evaluation. However, it highlighted the fact that the suite does not run out of the box with the compilers tested.

It is also worth noting, than in some cases, benchmarks are only available for one programming language. For instance, the EPCC OpenACC benchmark suite has tests exclusively in C, whereas other suites like SPEC ACCEL OpenACC, attempt to include cases that use both C, Fortran, and a combination of both. Furthermore, the majority of tests in the SPEC ACCEL OpenACC benchmark use the *acc kernels* directive which is not well supported in GCC. In real applications, *OpenACC parallel* is used because it is more prescriptive than *kernels* and more tunable for a target architecture. One suggestion would be to encourage benchmark maintainers, like SPEC HPG, to revise benchmarks so they give preference to the more commonly used *acc parallel* directive.

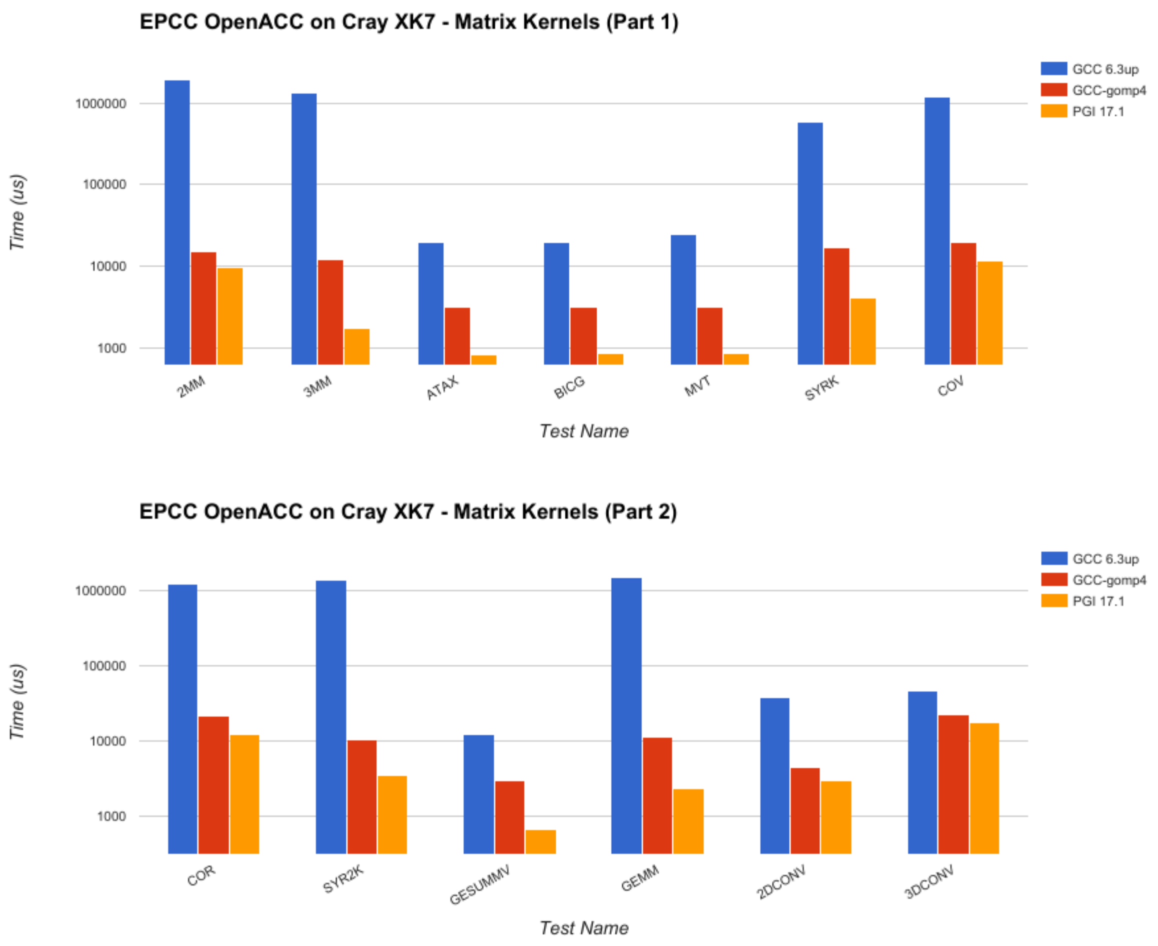

Figure 5: EPCC OpenACC benchmark: Execution time of matrix kernels on the Cray XK7 system.

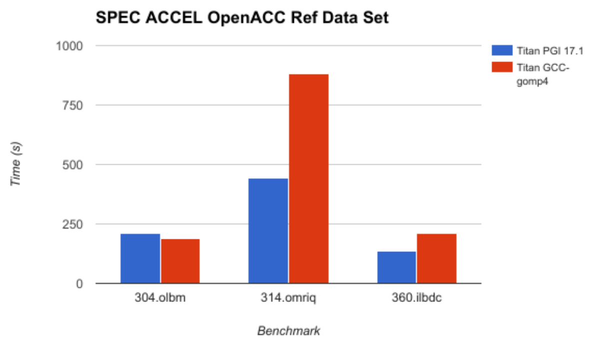

Figure 6: SPEC ACCEL OpenACC: Measured estimates for eference data sets.

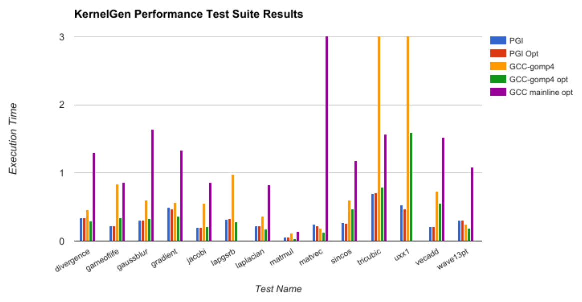

Figure 7: KernelGen Performance Comparison.

VII. Conclusion

This study presents an overview of the design and implementation of GCC's OpenACC compiler. The latest version of the Mentor Graphics public branch of GCC, namely GCC-gomp4, was used to evaluate the performance and correctness of the implementation in its current state. The paper details the known limitations of the compiler, which in part determined the experiments choices made along the way, as well as how the compiler maps certain commonly used OpenACC constructs onto the hardware.

The results show that the GCC-gomp4 compiler can in some cases outperform mature OpenACC compilers like PGI, as observed in the case of the 304.olbm benchmark for the ref data sets. Overall, for the SPEC ACCEL OpenACC benchmarks used, for example, the GCC-gomp4 compiler is on average approximately 47% slower than PGI for the ref data set. Due to the fact that some OpenACC features are not yet implemented in GCC, the set of tests that can be executed without modification is limited.

Having additional compilers that support OpenACC is an important requirement in order for developers to continue adopting OpenACC. In addition, it is important to note that the different architectural trends in HPC, make it a requirement for compilers to be able to support more than one target. The PGI compiler already provides the option to build OpenACC code for manycore architectures in addition to GPUs, and our results on Percival showed that it performs well on both architectures. For the comparison to be fair, support for more architectures is needed in GCC and other OpenACC implementations.

Another important finding from this work is that OpenACC test suites do not currently help to verify and validate the latest OpenACC specification. Several of the benchmarks utilized here have not been updated since OpenACC v1.0 which further limits researchers' ability to comprehensively evaluate OpenACC compilers. Additional work is needed to ensure that benchmarks are updated to use the latest features and also to keep up with changes to the semantics of the specification. One such subtle change in the behavior of the `copy` directive occurred between version 2.0 and 2.5. Such changes need to be tracked closely and reflected in the test suites.

VIII. Future Work

As work on the GCC-gomp4 branch progresses and changes are included in a future GCC release, this study should be repeated to include benchmarking of a wider range of features in OpenACC. This study can also be expanded to perform a comprehensive evaluation and comparison of current OpenACC compilers.

In addition, tests used in this evaluation should be expanded to include features introduced in the latest version of the OpenACC specification.

Acknowledgment

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] C. Wang, R. Xu, S. Chandrasekaran, B. Chapman, and O. Hernandez, "A validation testsuite for openacc 1.0," in *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, May 2014, pp. 1407–1416.

[2] N. Johnson, "EPCC OpenACC benchmark suite," https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openacc-benchmark-suite, 2013.

[3] "SPEC ACCEL Benchmark Suite," https://www.spec.org/accel/.

[4] "The kernelgen performance test suite," https://hpcforge.org/plugins/mediawiki/wiki/kernelgen/index.php/Performance_Test_Suite.

[5] "OLCF Lends Expertise for Introducing GPU Accelerator Programming to Popular Linux GCC Compiler," https://www.olcf.ornl.gov/2013/11/14/olcf-lends-expertise-for-introducing-gpu-accelerator-programming-to-popular-linux-gcc-compiler/.

[6] "GCC OpenACC Wiki," https://gcc.gnu.org/wiki/OpenACC.

[7] "GCC OpenACC Wiki: Loop Partitioning," https://gcc.gnu.org/wiki/OpenACC#Automatic_Loop_Partitioning.

[8] W. Joubert, R. K. Archibald, M. A. Berrill, W. M. Brown, M. Eisenbach, R. Grout, J. Larkin, J. Levesque, B. Messer, M. R. Norman, and et al., "Accelerated application development: The ORNL Titan experience," *Computers and Electrical Engineering*, vol. 46, May 2015.

[9] L. Grillo, F. de Sande, and R. Reyes, "Performance evaluation of openacc compilers," in *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*. IEEE, 2014, pp. 656–663.

[10] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The NAS parallel benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[11] "SPEC OMP2012 Benchmark Suite," https://www.spec.org/omp2012/.

[12] "SPEC ACCEL Benchmark Suite Documentation," https://www.spec.org/auto/accel/Docs.

[13] "Parboil Benchmarks," http://impact.crhc.illinois.edu/parboil/parboil.aspx.

[14] D. Mikushin, N. Likhogrud, E. Z. Zhang, and C. Bergström, "Kernelgen – the design and implementation of a next generation compiler platform for accelerating numerical models on gpus," in *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, ser. IPDPSW '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 1011–1020. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2014.115

[15] M. Christen, O. Schenk, and Y. Cui, “Patus for convenient high-performance stencils: Evaluation in earthquake simulations,” in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, Nov 2012, pp. 1–10.

[16] “Performance test suite for openacc compiler, intel mic, patus and single-core cpu,” https://github.com/pathscale/OpenACC-benchmarks.

[17] G. Juckeland, O. Hernandez, A. C. Jacob, D. Neilson, V. G. V. Larrea, S. Wienke, A. Bobyr, W. C. Brantley, S. Chandrasekaran, M. Colgrove, A. Grund, R. Henschel, W. Joubert, M. S. Müller, D. Raddatz, P. Shelepugin, B. Whitney, B. Wang, and K. Kumaran, *From Describing to Prescribing Parallelism: Translating the SPEC ACCEL OpenACC Suite to OpenMP Target Directives*. Cham: Springer International Publishing, 2016, pp. 470–488. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-46079-6_33