A VALIDATION SUITE FOR HIGH-LEVEL
DIRECTIVE-BASED PROGRAMMING MODEL FOR
ACCELERATORS

A Thesis
Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Jinxin Yang
May 2015
A VALIDATION SUITE FOR HIGH-LEVEL
DIRECTIVE-BASED PROGRAMMING MODEL FOR
ACCELERATORS

Jinxin Yang

APPROVED:

Dr. Barbara Chapman, Chairman
Dept. of Computer Science

Dr. Guoning Chen
Dept. of Computer Science

Dr. Deniz Gurkan
Dept. of Engineering Technology

Dean, College of Natural Sciences and Mathematics
Acknowledgement

It took me long enough to get this done, but it would have taken forever without help, both in the form of encouragement and emotional support, and in the form of other people physically performing tasks that I would otherwise have had to do.

I would like to express my gratitude to my advisor, Dr. Barbara Chapman for providing an excellent environment for research. It was an absolute privilege to work under her tutelage, inspiring me to explore ideas for prospective research topics, obtaining constructive feedback on my performance, and providing me the opportunity to attend conferences in order to interact, gather, and share ideas with fellow members of the HPC (High Performance Computing) community.

I would also like to thank my mentor, Dr. Sunita Chandrasekaran. Without her continual support, guidance, and patience, this project would not have been possible. I would like to extend a special word of thanks to Dr. Abid Malik and Deepak Eachempati for their guidance and support in my previous project.

And much thanks to everyone who has been supportive and patient towards me over the last two years, especially my family, my lab mates, my friends, my dog Ellie and cat Sugar. Most of all to my girlfriend, who is amazingly patient and supportive.
A VALIDATION SUITE FOR HIGH-LEVEL DIRECTIVE-BASED PROGRAMMING MODEL FOR ACCELERATORS

An Abstract of a Thesis
Presented to
the Faculty of the Department of Computer Science
University of Houston

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Jinxin Yang
May 2015
Abstract

The broad adoption of accelerators boosts the interest in accelerator programming models. OpenACC is an emerging and directive-based programming model for accelerators that typically enables non-expert programmers to achieve portable and productive performance of their applications. The model is gaining popularity and being used for accelerating many types of applications, ranging from molecular dynamics codes to particle physics models. However, in order to ensure correctness of OpenACC’s compiler implementation and determine its conformance to the specification, there is a critical requirement of an up-to-date validation suite.

In this thesis, we present a portable and robust validation suite execution environment to serve this purpose. The validation suite consists of a scalable testing infrastructure and more than 140 test cases for the most recent OpenACC 2.0 programming model, both for C and FORTRAN languages. The test cases aims to identify and resolve ambiguities within the OpenACC 2.0 specification. The framework of this testsuite is also robust enough to create test cases for the future releases.

We evaluate three commercial OpenACC compilers that are being widely used for porting applications to accelerators and assist them in identifying and resolving compiler bugs helping them improve the quality of their compilers. We test their five stable compiler versions and collect the pass rate for tracking the validating status. The results shows that the number of bugs decrease with every newer version of the compiler released demonstrating improved compiler quality. Based on the interesting observations during the development of the validation suite, we also give suggestions on improving the future specification, which are good feedback to the OpenACC community.
Contents

1 Introduction  1
   1.1 Heterogeneous Computing  2
   1.2 Motivation  3
   1.3 Method  5
   1.4 Contribution  7
   1.5 Organization  7

2 Background  9
   2.1 Modern GPU Architecture  10
   2.2 Programming Models for GPU  13
      2.2.1 CUDA  13
      2.2.2 OpenCL  15
      2.2.3 OpenACC  17
   2.3 Compiler Support for GPU Programming  18

3 Related Work  20
   3.1 Language Conformance Checking  21
   3.2 Random Program Generator  21
   3.3 LLVM Testing Infrastructure  22

4 OpenACC Application Programming Interface  24
## List of Figures

2.1 Internal structure of a modern GPU with eight 16-wide SIMD processing units combined with six 64-bit memory interfaces .......................... 10

2.2 Memory Bandwidth for the CPU and GPU ........................................ 12

2.3 CUDA Heterogeneous Programming Execution Model .......................... 14

2.4 OpenCL Platform Architecture .......................................................... 16

4.1 A typical accelerator that supports three levels of parallelism ............... 27

4.2 OpenACC execution model ............................................................... 28

4.3 Structured vs Unstructured data lifetimes ......................................... 37

5.1 enter data create clause ................................................................. 44

5.2 Framework of the test infrastructure (in an example of enter_data_if) 48

6.1 Test pass rate for PGI compiler and their versions ............................. 79

6.2 Test pass rate for Cray compiler and their versions ............................ 79

6.3 Test pass rate for CAPS compiler and their versions .......................... 80
## List of Tables

6.1 Experimental results on three compilers, part 1 . . . . . . . . . . . . . . . 71
6.2 Experimental results on three compilers, part 2 . . . . . . . . . . . . . . . 72
6.3 Experimental results on three compilers, part 3 . . . . . . . . . . . . . . . 73
6.4 Experimental results on three compilers, part 4 . . . . . . . . . . . . . . . 74
6.5 Experimental results of C atomic clauses . . . . . . . . . . . . . . . . . 75
6.6 Experimental results of FORTRAN atomic clauses . . . . . . . . . . . . . 76
6.7 Available Corner Test Results . . . . . . . . . . . . . . . . . . . . . . . . . 77
6.8 Bugs identified in different compilers . . . . . . . . . . . . . . . . . . . . . 81
Chapter 1

Introduction

Recent years have seen a rise of massively-parallel supercomputing systems that are based on heterogeneous architectures combining multi-core CPUs with many-thread accelerators. OpenACC is directive-based, high-level programming model for accelerators. It allows the users to insert non-executable pragmas and guide the compiler to handle the low-level complexities of the system. However, different compiler developers may interpret a given specification differently leading to more than one way of implementing a given construct. Our validation suite for OpenACC 2.0 is to help to validate compiler implementation of programming models, locate the ambiguities in the OpenACC 2.0 specification and provide an adequate feedback to the compiler vendors.
1.1 Heterogeneous Computing

“Heterogeneous Computing” refers to the systems that use more than one kind of processor [21]. These are multi-core systems that gain the performance not just by adding cores, but also by incorporating specialized processing capabilities to handle particular tasks. Heterogeneous systems utilize multiple processor types, typically CPUs with General-Purpose Graphic Processing Units (GPGPUs) [37]. GPU processing, apart from its well-known 3D graphics rendering capabilities, can also perform mathematically intensive computations on very large data sets, while CPUs can run the operating system which will handle and perform traditional serial tasks.

Since the end of 2010, nearly all the new desktop computers have been equipped with multi-core processors, with dual-core and even quad-core processors that have entered the mainstream of affordable computing. A current example is the recent Intel Core i7 microprocessor [24] with four processor cores, each of them is an out-of-order, multiple-instruction issue processor implementing the full x86 instruction set, supporting hyper-threading with two hardware threads, designed to maximize the execution speed of sequential programs. However, multi-core processing also posed some challenges of its own. The extra cores required to fuel their instruction pipelines come at a cost of both increased processor size and high power consumption, which becomes the main bottleneck of the heterogeneous computing.

Meanwhile, the design and development of many-thread processors (especially on GPUs) show interesting progress, which are growing in both sophistication and complexity domains, spurred on by advances in semiconductor technology [30]. GPUs
focuses more on the execution throughput of parallel applications. The many-thread processors began with a large number of threads, and once again, the number of threads increases with each generation. Another example is the NVIDIA GTX680 [16] graphics processing unit (GPU) with 12,288 threads, executing in a large number of simple, in-order pipelines. GPUs have vector processing capabilities that enable them to perform parallel operations on very large sets of data and to do it at much lower power consumption than that on CPUs. Besides the ability to improve the 3D graphics performance by offloading graphics from the CPU, they become increasingly attractive for more general purposes now, such as addressing data parallel programming tasks. Since 2003, GPUs have led the race of the floating-point performance. As of 2012, the ratio of peak floating-point calculation throughput between many-thread GPUs and multi-core CPUs is more than 10.

Moreover, due to the performance constraints on power and scalability in multicore CPUs development, software and systems designers increasingly look into the vector processing capabilities of modern GPUs.

1.2 Motivation

OpenACC [18] has gained its popularity since 2011 from modern open-source or commercial compilers, e.g., GNU [12], PGI [14], CAPS [25] and Cray [2] compilers. However, different compiler groups or designers may have various interpretation for a given specification. This leads to more than one way of implementing a given
construct. One of the reasons in the specifications can be ambiguities. The implementation of this OpenACC validation suite is based on the construction of another construct that is similar but not identical. Therefore it is not strange to find such situations especially when there is evolution of specification and implementation of a standard.

As a motivational example, the OpenACC execution model describes gang, worker and vector clauses to specify different levels of parallelism on accelerators. The usual way of occurrence is in order hence choosing different values in the clause may have a substantial effect on the performance of the application. For instance, they may correspond to the block, warp, and threads respectively from the perspective of the CUDA programming model. Mapping is however the implementation-dependent and is capable of creating several possible combinations.

Take one of the new features in the OpenACC 2.0 [11] specification—Nested Parallelism as another example. Listing 1.1 shows a simple demonstration of this feature: parallel construct within kernels construct. One may come up with several questions about the behaviours of this code snippet:

- Can two compute regions be tightly nested?
- Are n threads launching 10 kernels? Or is the outer level sequential?
- What about the race condition on the c array?

The specification does not state the expected model of this “parallel within kernels”
scenario or *vice versa*. When the code was executed, there were different results observed that were generated by different compilers which led to inconsistency in the behavior of the compilers. This can also widen the performance gap while comparing different compiler results in deep testing.

```c
/* Matrix-vector multiplication on NVIDIA GPU */
#pragma acc kernels copyin(a[0:n*n], b[0:n]) copy(c[0:n])
{
#pragma acc parallel num_gangs(10)
#pragma acc loop gang
  for(int i=0; i<n; i++) {
    real_t temp=0.0;
#pragma acc loop reduction(+:temp)
    for(int j=0; j<n; j++) {
      temp += a[i*n+j] * b[j];
    }
    c[i] = temp;
  }
}
```

Listing 1.1: A ambiguity on the Nested Parallelism feature

### 1.3 Method

We propose an OpenACC compiler validation test suite and infrastructure that is used to validate the conformance, correctness and completeness of different compiler implementations. The test suite consists of two parts: the test codes and the test infrastructure. There is assurance that each test case test for a single OpenACC feature only for the production of high quality results. (on occasion, some features
have dependency on others, e.g., parallel/kernels default).

The test codes include uniform and main driver function and test module functions. The uniform main driver is almost the same for each test except the feature information (e.g. name) and main function (both for C and FORTRAN) lies within. The test module is written following an XML syntax structure that pairs the OpenACC directive/clause to be tested (e.g. `<test> async </test>`). The variable between XML tags is the target feature we want to test or the cross test.

A bash script will then be used to parse the test module and automatically generate two associated test codes: regular and cross tests. The use of template-based tests have several advantages. First of all, it only needs minimal effort to develop the completed test code. As a result, developers only need to focus on designing the test modules instead of writing the entire test programs including the main function and input/output every time. The generated test code is a standalone C/FORTRAN code, i.e., it can be compiled by any OpenACC compiler.

Finally, a test harness will then compile the program, run the executable, check for the results and generate reports. The harness also supports single or batch test (directory based). A bug report is generated which qualitatively and quantitatively analyzes features that are being tested. This infrastructure is extensible enough to accommodate new features as the specification evolves.
1.4 Contribution

We create the entire OpenACC 2.0 validation suite which now serves the OpenACC community members:

- We check the conformance, correctness and completeness of certain compilers for the OpenACC 2.0 new features. The tests ensure the width and depth of all the directives. We also minimize the amounts of programming to add new feature test in the OpenACC future version.

- We help the compiler developers to identify implementation bugs, give suggestions on the correct behaviours and deliver detailed phase report.

- We clarify several ambiguities in the OpenACC 2.0 specification and report to the OpenACC community for improving the evolving OpenACC specification.

- We demonstrate the testsuite as a good and rigorous tutorial for the new OpenACC users and compiler developers.

1.5 Organization

This thesis is organized into the following sections as described below:

- Chapter 2 provides the background on modern GPU architecture and its programming models.

- Chapter 3 discusses some of the related works to this effort.
• Chapter 4 briefly gives an overview of the OpenACC programming model and the feature set.

• Chapter 5 explains the design of the validation testsuite and demonstrates several typical test cases.

• Chapter 6 elaborates how our validation suite evaluates the OpenACC compilers and the results for three vendor compilers.

• Chapter 7 contains our conclusion and future work.
Chapter 2

Background

GPUs have undergone at a fast-paced rate of change in recent years. The design philosophy of GPU architecture has been shaped by the fast-growing video game industry that exerts tremendous economic pressure for the ability to perform a massive number of floating-point calculations. Programming models like CUDA, OpenCL, OpenACC and OpenMP accelerator extensions, also have been created to address the programmability challenge of modern GPUs. They either give programmers explicit control of these parallel programming details or provide useful hints to compilers to perform certain transformation and optimizations on the annotated code region. Therefore compiler support for these programming models plays critical role in the evolution of both software and hardware platform development of GPUs.
2.1 Modern GPU Architecture

Figure 2.1 shows a block diagram of the commercial NVIDIA GeForce 8800GTX GPU, which is a typical CUDA-capable GPU [23]. It is organized into an array of highly threaded streaming multiprocessors (SMs). Normally, two SMs form a building block, however, the number of SMs in a building block can vary from one generation of CUDA GPUs to another generation. SMs contain personal register files within 16 load or store components as well as 4 Special Function Units (SPUs) that operate functions like \texttt{sine} and \texttt{cosine}. Also, in Figure 2.1, each SM has a number of streaming processors (SPs) that share control logic and an instruction cache.

![Figure 2.1: Internal structure of a modern GPU with eight 16-wide SIMD processing units combined with six 64-bit memory interfaces](image)

The more current chip, is manufactured by NVIDIA around the Kepler architecture has moved towards a more general purpose design. This has specified SMs that
are being increased in size and consist of double warp instruction now. Such as the NVIDIA 7800 chip released in 2005 which is worthy to be noticed of its computing functionality that is being segmented in varied formats of processing components with the chip design predating the unified shader model. On the other hand, the NVIDIA 8800GTX chip bears all variation among processors that have been replaced with a single format of those “streaming processor”, as we discussed above.

Each current GPU equips with multiple gigabytes of Graphic Double Data Rate (GDDR) DRAM, referred to as global memory. These GDDR DRAMs differ from the system DRAMs on the CPU motherboard in that they are the frame buffer memory that is used for graphics. For those graphics applications, they hold video images and texture information for 3D rendering. For computing however, they work as very high bandwidth off-chip memory, though with somewhat longer latency than typical system memory. Therefore for massively parallel applications, the higher bandwidth makes up for the longer latency. Figure 2.2 illustrates the significant memory bandwidth difference between CPUs and GPUs [4].

The GPU executes independently from the CPU, but is controlled by the CPU. Application programs running on the CPU use graphics API, runtime, and driver software components to communicate with the GPU. Most of the communication involves placing commands or data in memory buffers and transmitting them to the GPU. Graphical data that are accessed frequently (vertices, textures, output images) by the GPU are often placed in a high-bandwidth memory which is attached directly to the GPU, with the CPU being used to set the initial state for these objects. Even with the dedicated GPU memory, the CPU sends a great deal of data to the GPU
The NVIDIA G80 first introduced the CUDA architecture and had 86.4 GB/s of memory bandwidth with a communication link to the CPU cores via a PCI-Express Generation 2 (Gen2) interface. Over Gen2 interface, CUDA applications can transfer data from the CPU memory to the global memory at 4 GB/s, and the upload data back to the CPU memory at 4 GB/s. That is a combined total of 8 GB/s together. The communication bandwidth should also be expected to grow as the CPU bus bandwidth of the system memory that grows in the future.

With 12,288 threads, the GTX680 can exceed 1.58 teraflops in double precision. A good application typically runs 5,000–12,000 threads simultaneously on this chip. The level of parallelism that supports GPU hardware is increasing quickly. It
is therefore very important to work for high levels of parallelism when developing parallel-computing applications based on the modern GPU architectures.

### 2.2 Programming Models for GPU

In this section, we briefly introduce two low-level (CUDA and OpenCL) and one directive-based [35] high-level (OpenACC) programming models for GPU. The low-level programming models define language extensions and runtime APIs to allow programmers to manage parallelism and data delivery in accelerators, the directive-based high-level programming models provide the compiler automation and the runtime support for abstracting away many parallel programming details from programmers.

#### 2.2.1 CUDA

CUDA [40] stands for Compute Unified Device Architecture: It is an extension of the C, C++ and the FORTRAN programming language. It has been created by NVIDIA. Using CUDA allows the programmer to take advantage of the massive parallel computing power of GPUs in order to do the general purpose computation [45].

The CUDA programming paradigm is a combination of serial and parallel execution. Figure 2.3 shows an example of this execution model. The simple C code runs serially on CPU, also called the host. The kernel function expresses the parallel execution that is executed on a set of threads in parallel on the GPU; the GPU is
also known as the device. The kernel code is the CUDA code for a single thread. The number of thread blocks and threads within those blocks that execute the kernel in parallel are explicitly given upon calling of this function.

![CUDA Heterogeneous Programming Execution Model](image)

Figure 2.3: CUDA Heterogeneous Programming Execution Model

The serial code from CPU is the only one with the capacity to invoke the kernel function. The kernel configuration requires to be specified in order to call the kernel function. CUDA has a predefined data type \texttt{dim3} for the declaration of grid and thread blocks. The data type is a vector specifies the dimensions of the grid and thread blocks.

Three angular brackets are used in writing the call grid and block variables in
the kernel function, \(<\text{grid,block}>>\) as shown in Figure 2.3. Grids and thread blocks are created dynamically in this invocation. The value of the grid and block variables should be less than the maximum allowed sizes. The threads are scheduled in hardware rather than software. The kernel function usually consists of the return type \texttt{void}. It also consists of a \_\texttt{global}\_ qualifier which means such a version should be executed on the GPU.

NVIDIA is committed to supporting CUDA as hardware changes. As the CUDA evolves, it still encapsulates the hardware model, therefore developers do not need to worry about the hardware model changes [6]. For example, the legacy CUDA code running on a Fermi GPU should also work on a Kepler GPU. Learning the hardware and developing parallel algorithms are still difficult. But the infrastructure for writing, developing, debugging and maintaining the source code is straight forward and similar to the conventional serial programming.

2.2.2 OpenCL

OpenCL [19] is a standardized and cross-platform API designed to support portable parallel application development on heterogeneous computing systems. Like CUDA, OpenCL addresses the complex memory hierarchies and the data-parallel execution. It draws heavily on the CUDA driver API experience. Therefore people can always find the mappings of the OpenCL data parallelism model concepts, API calls, and memory types to their CUDA counterparts.

Figure 2.4 illustrates an overview of the OpenCL architecture [49]. A single
CPU-based “Host” controls multiple “Compute Devices” (for instance CPUs and GPUs are different compute devices). Each of these coarse grained compute devices are made up of multiple “Compute Units” (akin to execution units and arithmetic processing unit groups on multi-core CPUs) and within these are multiple “Processing Elements”. At the lowest level, these processing elements all execute OpenCL “Kernels”.

Figure 2.4: OpenCL Platform Architecture

OpenCL has a more sophisticated device management model that mirrors its support for multiplatform and multivendor portability [32]. OpenCL programs must be prepared to deal with much greater hardware diversity and thus will exhibit more complexity. The OpenCL model for device management, the OpenCL kernel compilation model and the OpenCL kernel launch are more complex in relation to their CUDA counterparts. However, the device compatibility drives OpenCL since it is more developed than NVIDIA’s CUDA. OpenCL is compatible with all programmable graphic processors unlike CUDA which supports NVIDIA GPUs only.
2.2.3 OpenACC

Unlike OpenCL and CUDA, OpenACC represents one of the high-level directive-based programming models. Directives are high level language constructs that programmers utilize in the provision of significant hints for compilers to perform specific optimizations and transformations on the region with the annotated code. The use of directives is expected to substantially improve the productivity of programming. Users still have the capability to attain high performance of their comparable to code written in OpenCL or CUDA in respect to the expectations that a critical choice of compiler and directives optimization strategies are in place. Choices of strategies vary with the types of accelerator [55].

The directive-based approach is highly dependent on the compiler for the generation of efficient codes for thread mapping and data layout. There are major challenges applicable for the extraction of optimal performance using other explicit programming models. However, the model simplifies programming done on heterogeneous systems hence saving on development time while preserving the original code structure which facilitates for portability. The device and the memory spaces on the host are separated from one another. The host has no capacity to access the device memory directly or *vice versa*.

Listing 2.1 shows a simple OpenACC parallel construct example. OpenACC allows users to specify three levels of parallelism in a data parallel region: coarse-grain parallelism “gang”, fine-grain parallelism “worker” and vector parallelism “vector”, to map to the multiple-level thread hierarchy of GPUs. Mapping these three-level
parallelism to the GPU threading structure will be left to the compiler and runtime systems, according to the hints given by the programmers. It can be a challenge for programmers, particularly on large programs with complex irregular data access pattern and thread synchronization.

```c
#pragma acc parallel num_gangs(256) num_workers(32) vector_length(32)
{
    #pragma acc loop gang
    for (int i=0; i<512; i++)
    #pragma acc loop worker
        for (int j=0; j<1024; j++)
    #pragma acc loop vector
        for (int k=0; k<2048; k++)
            foo(i,j,k);
}
```

Listing 2.1: A Simple OpenACC example

### 2.3 Compiler Support for GPU Programming

There are already a number of compilers that provide support for GPU programming.

Once, CUDA device functions and data qualifiers are added to a source file, it is no longer acceptable to a traditional C compiler. The code needs to be compiled by a compiler that recognizes and understands these additional declarations. Along with the introduction of CUDA architecture, NVIDIA provides its C compiler called NVCC [10]. PGI and Cray group later started to offer CUDA development toolkit within their new released compilers.
OpenACC has gotten more attention and support from compiler vendors in the last three years. Those include PGI, CAPS and Cray, open-source OpenACC compilers include accULL [44], GCC [12], OpenUH [48], OpenARC [34] and ROSE [43]. Chapter 4 will discuss more details on OpenACC programming model.

OpenACC as an emerging standard for GPU programming, its compiler support is critically important in terms of the understanding, interpretation and further implementation both for compiler groups and the specification itself. Therefore it is necessary to have a thorough conformance checking along with the evolving programming model.
Chapter 3

Related Work

Test suites are everywhere regardless of their scale and size, for nearly all kinds of real-world applications. From the simpliest micro-kernel program tests to large Operating System checking, we can never underestimate the importance of a test suite. To the best of our knowledge, we are the first team to develop a validation suite for validating implementations on OpenACC compilers. There are commercial or open source test suites which are primarily aimed to check for the conformance of specific language standards. Researchers also develop other related efforts that discuss ways and means to detect compiler bugs, which do not quite validate the compilers conformance to a specification. Open source compiler LLVM has its own testing infrastructure which is designed to trigger a specific bug in LLVM or as a way of the module test and the benchmarking LLVM performance.
3.1 Language Conformance Checking

Language conformance checking is necessary for the compiler development. There are existing commercial test suites, such as [20], [41] and [42] are primarily aimed to check for the conformance of C and Fortran standards. For the language specification, almost all the commercial or open source compilers have their own internal test suites for the language conformance checking.

For example, the GNU compiler’s test framework DejaGnu [46] is the collection of different and internal tests which include the checking for the language conformance, compiler flags, extension features and etc. Within DejaGnu, the language conformance takes a large proportion, such as for GCC, G++ and GFORTRAN compiler. Its purpose is to provide a single front end for all tests, therefore users are able to write tests for different kinds of programs. However, in contrast to DejaGnu, our test suite is mainly focusing on the functional test, which is more like a white-box test [31].

3.2 Random Program Generator

Csmith [56] is one approach that performs a randomized test-case generator hunting down compiler bugs using differential testing. The basic idea of randomized and 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 through a test harness.
Csmith aims to target correctness bugs instead of conformance bugs. They introduce semantic rules during their code generation process by using filter [28] functions, which allow or disallow certain productions depending on the context. This is reasonable when constructing a generator for a specific language, but very difficult for a directive-based approach as we are aiming for.

Therefore, Csmith’s approach is quite effective to detect 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.

### 3.3 LLVM Testing Infrastructure

The LLVM [33] compiler has its own testing infrastructure [9] which contains two major categories of tests: regression tests and whole programs.

The regression tests [27] are small pieces of code that test a specific feature of LLVM or trigger a specific bug in LLVM. The language, they are written, is in depends on the part of LLVM being tested. These tests are driven by the lit [8] testing tool (part of LLVM), which is a wrapper of executing instructions. Typically when a bug is found in LLVM, a regression test containing just enough code to reproduce the problem should be written and tested. For example, it can be a small piece of LLVM IR distilled from an actual application or a benchmark.

The whole programs are referred to as the “LLVM test suite” (or “test-suite”). They are pieces of the code which can be compiled and linked into a stand-alone
program that can be executed. These programs are generally written in high level languages such as C or C++. The whole programs are compiled using a user specified compiler and set of flags, and then executed to capture the program output and timing information. The output of these programs is compared to a reference output to ensure that the program is being compiled correctly. In addition to compiling and executing programs, the whole program tests serve as a way of benchmarking LLVM performance, both in terms of the efficiency of the programs generated, as well as the speed with which LLVM compiles, optimizes, and generates the code.

It is a way to write our test cases and use LLVM “test-suite” infrastructure as our driver to compile, execute and analyze. However we design our OpenACC test suite as an independent package for different compilers, a framework-free method (no dependency with other framework) will be better for us. For example, it decreases the portability of our testsuite if we also need to install LLVM infrastructure for every operating system that we are testing on.
Chapter 4

OpenACC Application Programming Interface

The OpenACC Application Programming Interface (API) provides a set of compiler directives, library routines, and environment variables that can be used to write the data-parallel FORTRAN, C, and C++ programs that run on accelerator devices, including GPUs. It is an extension to the host language. The OpenACC specification was initially developed by the Portland Group (PGI), Cray Inc., and NVIDIA, with the support from CAPS enterprise. Through introducing the execution and memory model, basic compute constructs and runtime libraries, we discuss how OpenACC programming model works on host and accelerator sides. We will walk through some concrete code examples to illustrate usage of some of the more commonly used OpenACC directives and APIs.
4.1 Introduction

OpenACC is an emerging and evolving standard for programming accelerator boards in conjunction with a host CPU, which can be a multi-core platform [51]. It is based on the usage of pragmas or directives that allow the application developers to mark regions of the code for acceleration in a vendor-neutral manner. It builds on top of prior efforts by several vendors (notably PGI, CAPS and Cray Enterprise) to provide parallel programming interface for heterogeneous systems, with a particular emphasis on platforms that are comprised of multi-core processors as well as GPUs. Among others, OpenACC is intended for the use on the nodes of large-scale platforms such as the Titan system at ORNL, where CPUs and NVIDIA GPUs are used to solve some of the nations’ most urgent scientific problems.

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 multi-core 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 the conditional execution, controls the number of threads, specifies the scope of the data that is 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 the host and accelerator is critical for the application performance on the emerging and 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 to 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 the 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 programming model gives great flexibility to the compiler implementation. For instance, different compilers can have a different interpretation of OpenACC three level parallelism: coarse grain parallelism “gang”, fine grain parallelism “worker” and vector parallelism “vector”. On a 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 Single Instruction Multiple Threads (SIMT, [36]) group of threads [55].

4.2 Execution Model

A general OpenACC target machine has a host and an attached accelerator device, such as a GPU. Most accelerator devices can support multiple levels of parallelism.
Figure 4.1 abstracts a typical accelerator that supports three levels of parallelism [38]. At the outermost coarse-grain level, there are multiple execution units. Within each execution unit, there are multiple threads. At the innermost level, each thread is capable of executing vector operations. Currently, OpenACC does not assume any synchronization capability on the accelerator, except for thread forking and joining. Once work is distributed among the execution units, they will execute in parallel from start to finish. Similarly, once work is distributed among the threads within an execution unit, the threads execute in parallel. Vector operations are executed in lockstep [32].

![Figure 4.1: A typical accelerator that supports three levels of parallelism](image)

Figure 4.2 shows how an OpenACC program starts its execution on the host using single-thread. When the host thread encounters a `parallel` or a `kernels` construct, a `parallel region` or a `kernels region`, that comprises all the code enclosed in the construct is created and launched on the accelerator device. The `parallel region` or `kernels region` can optionally execute asynchronously with the host thread.
and join with the host thread at a future synchronization point. The parallel region is executed entirely on the accelerator device. The kernels region may contain a sequence of kernels, each of which is executed on the accelerator device.

The kernel execution follows a fork-join model [38]. A group of gangs are used to execute each kernel. From the perspective of the programmer working with a parallel or kernels construct, the OpenACC execution model has three levels: gang, worker and vector. How these constructs map to the underlying hardware depends on the device capabilities and what the compiler thinks is the best mapping for the problem [17]. A group of workers can be forked to execute a parallel work-sharing loop that belongs to a gang. Typically a gang executes on one execution unit, and a worker runs on one thread within an execution unit. The programmer can
instruct how the work within a parallel region or a kernels region is to be distributed among the different levels of parallelism on the accelerator.

4.3 Memory Model

In an OpenACC memory model, the host memory and the device memory acts as a separate memory space. The host is assumed not to be able to access the memory of the device directly while being unable to access host memory directly. This is to verify compatibility of the OpenACC programming model to a range of accelerator devices. This includes most of the GPUs that lack capacity of unified memory access between CPUs and GPUs. The GPUDirect [47] and the unified virtual addressing introduced by NVIDIA in CUDA 4.0 allow a single virtual address for both device memory and host memory and allow direct cross-device memory access between distinct GPUs. However, there is still no possibility for device memory and cross-host.

The concept of the separate host and accelerator memories is very apparent in the low-level accelerator programming languages such as CUDA or OpenCL, in which the data movement between the memories can dominate the user code. In the OpenACC model, the data movement between the memories can be implicit and managed by the compiler, based on directives from the programmer [11]. However, the programmer must be aware of the potentially separate memories for many reasons, including but not limited to:

- Memory bandwidth between the device memory and host memory determines
the level of compute intensity which is needed for the effective acceleration of
the given region of code, and

• Offloading may be prohibited by the limited device memory size which operates
  on very large amounts of data.

• Host addresses are only valid to the host and stored to pointers on the host;
  addresses stored to pointers on the device have the capacity to be only valid
  on the host. Dereferencing of host pointers on the device or on the host may
  be invalid on such targets.

Just like in CUDA C/C++, in OpenACC input data needs to be transferred
from the host to the device before kernel launches and result data needs to be trans-ferred back from the device to the host [32]. However, unlike in CUDA C/C++
where programmers need to explicitly code the data movement through API calls, in
OpenACC they can just annotate which memory objects need to be transferred, as
shown by line 2 in Listing 4.1. The OpenACC compiler will automatically generate
the code for memory allocation, copying, and de-allocation.
// Compute matrix multiplication.
#pragma acc kernels copyin(a,b) copy(c)
for (i = 0; i < SIZE; ++i) {
  for (j = 0; j < SIZE; ++j) {
    for (k = 0; k < SIZE; ++k) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}

Listing 4.1: A Simple OpenACC data copy example

OpenACC employs a fairly weak consistency model on the accelerator device for memory. Other than data on the accelerator being sharable to all execution units, OpenACC fails to provide a reliable approach for allowing a single execution unit to consume data produced by other execution units. Two major reasons are posed behind this. First, OpenACC does not provide any mechanism for synchronization between execution units [52]. Second, memories between different execution units are not coherent. Although some hardware provides instructions to explicitly invalidate and update cache [32], they are not exposed at the OpenACC level. Therefore, different execution units are required to work on disjoint memory sets in OpenACC. Threads within an execution unit can also share memory and threads have coherent memory. However, OpenACC currently only mandates a memory fence at the thread fork and join, which are also the only synchronizations OpenACC provides for threads [26]. While the device memory model may appear very limiting, it is not in practice. For data-race free [29] OpenACC data-parallel applications, the weak memory model works quite well.
4.4 Basic OpenACC Features

This section will dive into some details of how one can write real OpenACC programs by understanding the basic features.

4.4.1 Parallel and Kernels Constructs

The parallel and kernels constructs (Listing 4.2) are two constructs that can be used to specify which part of the program is to be executed on the accelerator.

```
#pragma acc parallel num_gangs(100), vector_length(128)
{
#pragma acc loop gang, vector
    for(int i = 0; i < n; i++)
        y[i] += a*x[i];
}
#pragma acc kernels loop
for(int i = 0; i < n; i++)
    y[i] += a*x[i];
```

Listing 4.2: Parallel and Kernels Constructs

When a program encounters a parallel construct, the execution of the code within the structured block of the construct (also called a parallel region) is moved to the accelerator. Gangs of workers on the accelerator are created to execute the parallel region, as shown in Figure 4.2. Initially only one worker (the “gang lead”)
within each gang will execute the parallel region. The other workers are conceptually idle at this point. They will be deployed when there is more parallel work at an inner level. The number of gangs can be specified by the \texttt{num\_gangs} clause, and the number of workers within each gang can be specified by the \texttt{num\_workers} clause.

```c
#pragma acc kernels
{
    #pragma acc loop num\_gangs(512)
    for (int i=0; i<1024; i++)
        b[i] = a[i];

    #pragma acc loop num\_gangs(256)
    for (int j=0; j<1024; j++)
        c[j] = b[j]*4;

    for (int k=0; k<1024; k++)
        d[k] = c[k]*2;
}
```

Listing 4.3: A kernels region breaks into several kernels

A \textit{kernel region} is may consist of several kernels where each utilizes different number of gangs, a different vector lengths and a different number of workers. Therefore, there is no \texttt{num\_gangs}, \texttt{num\_workers}, or \texttt{vector\_length} clause on the kernels construct. The kernels region is also capable of being divided into different kinds of kernels (Listing 4.3) one for each loop and which will be executed on the accelerator in order. It is also possible that some implementations may decide not to generate a kernel for the \texttt{k} loop and therefore this kernels region will contain two kernels—one for the \texttt{i} loop and the \texttt{j} loop each and the \texttt{k} loop is executed on the host.
The kernels construct and the parallel construct are designed from two distinct views. Kernels construct is characterized by substantial description. The intentions of the programmer are described. Mapping and partitioning of the program to the hardware underuse is by the care of the compiler. On the other hand, the parallel construct is more prescriptive. The compiler works as per the instructions of the programmer. The programmer has much power in identification of locations to generate kernels and the approaches of parallelizing and scheduling loops.

4.4.2 Loop Construct

The significance of the parallel and kernels constructs will be visible when it is used with the \texttt{loop} construct. The loop directive is capable of describing the type of parallelism that is suitable for use in the execution of the loop. It also declares loop private variables and arrays and reduction operations. The listing 4.4 indicates the basic utilization of the three levels of parallelism (gang, worker and vector) via loop construct.
Listing 4.4: Gang Worker and Vector example

A gang loop construct is always associated with a loop. The gang loop construct is a work-sharing construct. The compiler and the runtime will make sure that the iterations of a gang loop are shared among all gang leads encountered in the loop construct. In Listing 4.4, because 256 gang leads will encounter the loop construct, each lead will be assigned two iterations. Now the execution of the parallel loop will be more efficient and likely to achieve speedup.

The worker loop construct is also a work-sharing construct. In Listing 4.4, the 32 workers in a gang will work collectively on the 1024 iterations of the $j$ loop in each of the two iterations of the $i$ loop assigned to the gang.

The vector clause on a loop construct is often used to express the innermost vector or SIMD (single instruction, multiple data) mode loop in an accelerator region.

On a GPU, a possible implementation is to map a gang to a CUDA block, a worker to a CUDA warp, and a vector element to a thread within a warp. However,
this is not mandated by the OpenACC specification and an implementation (compiler/runtime) may choose a different mapping that is based on the code pattern within an accelerator region for the best performance.

### 4.4.3 Data Management

In OpenACC, the host memory and device memory are separated. Data transfer between the host and the accelerator can play a significant role in the overall performance of an OpenACC application. For example, when a computationally intense loop nest of an iterative solver, implemented using a parallel loop, transfers data back and forth between the host and the accelerator at every iteration, then there may be a loss of performance. The OpenACC `data` and `enter/exit data` (as one of the new features in the OpenACC 2.0 specification) constructs allow one to exploit and reuse by avoiding data transfers during multiple executions of parallel or kernels regions.

Previous example like Listing 4.1 shows `copy`, `copyin`, and `copyout` clauses that are used on parallel or kernels constructs. These are called data clauses. A data clause has a list of arguments that are separated by a comma. Each argument can be a variable name or a subarray specification. The OpenACC compiler and the runtime will create a copy of the variable or the subarray in the device memory. Reference to the variable or the subarray within the parallel or kernels constructs will be made to the device copy.

OpenACC uses the data construct to allow the programmer to manage data
lifetimes on the device. This promotes structured programming, but not all data lifetimes are easily amenable to structured lifetimes. OpenACC 2.0 adds two new directives, “enter data” and “exit data”, which act like the beginning and the end of a data region. Figure 4.3 shows the difference between structured and unstructured data lifetimes, and if there is no exit data directive, the data lifetime continues until the end of the program.

```c
#pragma acc data copyin(a[0:n]) create(b[0:n])
{
    ...
}

#pragma acc enter data copyin(a[0:n]) create(b[0:n])
...
#pragma acc exit data delete(a[0:n])
...
#pragma acc exit data copyout(b[0:n])
```

Figure 4.3: Structured vs Unstructured data lifetimes

### 4.4.4 Procedure Calls

In OpenACC 1.0, procedure calls are very limited. Because the GPU devices has no linker, separate compilation was not supported. In fact, the NVIDIA Tesla [15] (compute capability 1.x) devices does not have a mechanism for procedure calls, so all procedure calls had to be inlined, which is an inefficient design.

Procedures and the separation compilation are one of the fundamental features of high level programming; they allow for libraries, modularity, and provide so many benefits that it is hard to justify a programming strategy without real procedure
calls. Enabled by CUDA 5.0 [5] features, OpenACC adds support for procedure calls. It is not quite as simple as just removing the restriction. Procedure may have “orphaned” loop directives. The compiler has to know whether that loop directive is to be mapped across gangs, workers, or vector lanes. Also, the compiler has to know what procedures are needed on the device. Therefore, routine directive is created for OpenACC 2.0.

```c
#pragma acc routine worker
extern void matvec(float* v, float* x, ...);

#pragma acc parallel loop num_gangs(200)
for( int i = 0; i < n; ++i ){
    v[i] += rhs[i];
    matvec( v, x, a, i, n ); // procedure call on the device
}

// in another file
#pragma acc routine worker
void matvec(float* v, float* x,
            float* a, int i, int n ){
    float xx = 0;
    #pragma acc loop reduction(+:xx)
    for( int j = 0; j < n; ++j )
        xx += a[i*n+j]*v[j];
    x[i] = xx;
}
```

Listing 4.5: routine directive in OpenACC 2.0

Listing 4.5 gives a simple example of using this directive. In this case, the routine directive appears just above the prototype for the procedure. This tells the compiler
that there will be a device copy of the routine \texttt{matvec}. It also tells the compiler that there may be a loop in \texttt{matvec} that targets the worker-level parallelism, so calls to \texttt{matvec} may not appear within a worker loop. Essentially, it says that worker-level parallelism has to be reserved to the routine \texttt{matvec}, and may not be used by the caller. The compiler will generate the code twice for this routine, once for the host and again for the accelerator; when generating the code for the accelerator, it will split the iterations of the j loop across the workers within a gang.

4.4.5 Asynchronous Behaviour

OpenACC provides support for asynchronous computation and data transfer. An \texttt{async} clause can be added to \texttt{parallel}, \texttt{kernels}, or \texttt{update} directive to enable asynchronous execution. If there is no \texttt{async} clause, the host process will wait until the parallel region, kernels region, or updates are complete before continuing. If there is an \texttt{async} clause, the host process will continue with the code following the directive when the parallel region, kernels region, or updates are processed asynchronously. An asynchronous event can wait using the wait directive or OpenACC runtime library routines.
Typically, an OpenACC program can have a number of async queues by adding a value expression to the `async` clause (Listing 4.6). Asynchronous activities with the same async value will be executed that they are enqueued by the host thread. Activities with different async value can execute in any order. Async queues correspond roughly to CUDA streams.

### 4.4.6 Runtime Library

OpenACC provides a richer set of routines to achieve the same effect as using directives. Here we simply list some new API routines for OpenACC 2.0.

- `acc_copyin`, `acc_create`, `acc_copyout` and `acc_delete`: act like the relevant data clauses, more or less like a `enter data` or exit data directive.

- `acc_is_present`: tests whether all the given data is already present on the device.
- `acc_update_device` and `acc_update_local`: act like the corresponding `update` directive.

- `acc_deviceptr`: returns the device pointer corresponding to the given host address; this can be used, for instance, to pass to an explicit CUDA kernel.

- `acc_hostptr`: does the inverse, returns the host address corresponding to a device address.

- `acc_map_data`: takes some device address that was directly allocated and adds the entry to the OpenACC runtime `present` table that this device address corresponds to the given host address.

- `acc_unmap_data`: undoes map’s effect.
Chapter 5

OpenACC Validation Suite

OpenACC 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 chapter, we present a robust and scalable testing infrastructure that serves this purpose. During the development, we try to find the minimal testing unit first and then cover as many features as possible. We also design the test module as a way that can be further applied with cross and corner tests. These design principles help to ensure the width and depth of all the tests. This entire validation suite consists of more than 140 test cases for all the new directives and clauses of the OpenACC 2.0 specification, both for C and FORTRAN languages. 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 associated clauses, that are being tested, have been implemented correctly.
5.1 Suite Infrastructure Design

We develop this testsuite infrastructure based on our previous suite [39], [50] and [51] for the OpenACC 1.0. According to the comments from three compiler vendors (PGI, Cray and CAPS) and our own working experience, we simplify the framework, increase the cover range, and add corner test features.

Similar to its first version, this testsuite will check if the directive passed or failed by verifying the result with a precalculated value. If the values do not match, it implies that there is an implementation issue. We define these tests as functional tests, which is a more general and high level test method derives from unit tests [57]. 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 example, Listing 5.1 (in Figure 5.1) shows a simplified test case for the `enter data create` clause. The `create` clause is used to declare that the variables, arrays, subarrays or common blocks need to be allocated ("created") in the device memory. Therefore in this case, the array `c` should be created once the program reaches the `enter data` directive. Within the innermost `j` loop, the product of array `a` and `b` stores in `temp` which finally updates the input array `c`. Since `create` clause must be used with the `enter data` directive here, while the functional test for the `create` feature may pass with the default attribute–`pcopy` for the array `c`, which will detect whether the array `c` is present in the device or copy `c` from the host memory. To gain more confidence of the test result, we designed a deeper test method, namely
the cross test. This test will help in validating “only” the directive or clause under consideration. The basic idea is that if we remove the directive that is being tested from the test code, the cross test should yield an “incorrect” result.

Listing 5.1: functional test

```
#pragma acc enter data 
 copyin(a[0:n*n], b[0:n]) 
 create(c[0:n])
```

```
#pragma acc kernels
#pragma acc loop gang independent
for(int i=0; i<n; i++) {
  real_t temp=0.0;
  #pragma acc loop reduction(+:temp)
  for(int j=0; j<n; j++) {
    temp+=a[i*n+j] * b[j];
  }
  c[i] = c[i] + temp;
}
```

Listing 5.2: cross test

```
#pragma acc enter data 
 copyin(a[0:n*n], b[0:n])
// create(c[0:n]) is removed
```

```
#pragma acc kernels
#pragma acc loop gang independent
for(int i=0; i<n; i++) {
  real_t temp=0.0;
  #pragma acc loop reduction(+:temp)
  for(int j=0; j<n; j++) {
    temp+=a[i*n+j] * b[j];
  }
  c[i] = c[i] + temp;
}
```

Figure 5.1: enter data create clause

Listing 5.2 shows the cross test for the `enter data create`, where `create` clause is simply removed. As a result, the array `c`, in the device, is passed from the host which is initialized with specific values and the cross test should get a different and updated `c` array from the functional test. However, if the result, in the cross test, is still the same as the functional test, it indicates that the directive that is being tested does not take any effect. The result will be reported and the functional test will be
redesigned. In some instances, simply by removing the directive that is 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.

In this OpenACC validation testsuite, we have designed more than 140 test cases (more than 300 if includes numerical operations, variable types and corner tests) covering the OpenACC C and FORTRAN feature set for the most recent OpenACC 2.0. These test cases cover tests for directives, clauses, runtime library routine, as well as environment variables. Each test has two versions: the functional test and the 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 5.2 shows the framework of the test infrastructure. A test template is written following an XML syntax structure that includes the OpenACC directive/clause to be tested. A bash 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 (some cases need other bash scripts for generating multiple operations, data types or corner tests).

As discussed above, first we perform the functional test. If the feature passes the test, the feature will need to undergo a deeper test, e.g., the cross test. Mostly, we
can create the cross test simply by removing the target directive or clause (here is
\texttt{if(flag)})) within the \texttt{<test>} tag. If the user wants to specifically write an unusual
cross test, the test case should be put into the \texttt{<cross>} tag. The \textit{Source Code}
Generator also creates the \textit{corner test} (maximum or minimum parameter) under the
same idea. If the feature does 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 that are being validated and the infra-
structure 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:

- \textbf{Compiler configuration}: User can set the configuration details (including
  the compiler being used, compilation flags, user-defined include path and li-
  brary path) for the compiler implementation to be validated.

- \textbf{Feature selection}: User can choose to test the directives, their clauses or any
  other feature of their choice at a given point of time. User can add features
  they want to validate to the test list.

- \textbf{Extensible test infrastructure}: We implemented two types of tests (e.g., func-
  tional and cross tests) for each of the features that are 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.
  For special test cases like \texttt{atomic} which needs to try various and numerical
operations, a script will replace the operator macro to generate different versions. 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, user is informed about the tests that passed or failed. After all the tests are executed, a full report will be generated demonstrating the result for each of the features in a form of prompt message or table. We append the bug reports with code snippets for vendors’ convenience, including the possible reasons of failure such as compilation error, incorrect results, execution error and so on.
Figure 5.2: Framework of the test infrastructure (in an example of `enter_data_if`)

```c
#include "acc_testsuite.h"
int test_acc_enter_data_if(FILE *log)
{
  // initialization
  ...  corner test case
}
int main()
{
  ...
  corner_test_acc_enter_data_if(logFile);
  ...
}
```
5.2 Single Test Design

As Figure 5.2 shows, the uniform and main driver will launch the test case C function (or FORTRAN subroutine) for both functional and cross test code. The key part of this testsuite is to properly design the functional test which should try to meet the following requirements:

- **Minimize the testing feature**: Not all the directives or clauses perform single function or take single type of parameter. For example, in the OpenACC 2.0 specification, the syntax of `loop gang` clause is `gang [ (gang-arg-list) ]`, where `gang-arg` is either `[num:] int-expr` or `static:size-expr`, and `size-expr` is one of `*` and `int-expr`. The test can be incomplete by writing only one single test function which includes all the sub-features, especially when execution halts because the middle-positioned feature fails. However, life can be much more easier if we separate tests into three dependent test functions: `[num:] int-expr, static:size-expr` for `*` and for `int-expr`.

- **Cover as many features as possible**: This is also the golden principle for one testsuite aiming for functional test. To achieve this, we need to understand the specification well and decompose as many features as possible. For each feature, it is also important to cover different data types (typically `int` and `double`), different `device_type` and take care of the corner conditions if possible. For example, compiler should throw a compilation error or adjust to a proper size if we give `num_gangs [ int-expr ]` clause a `-1`.

- **Make it easy for cross test**: Cross test is necessary for most of the cases.
By removing the target directive or clause, the cross test should always fail. If we can successfully minimize the testing feature as discussed above, the design for cross test will not be hard.

5.3 Test Cases Example

This section will discuss the design ideas of a few test cases written. (The rest of the tests mostly follows suit.)

5.3.1 Parallel and Kernels Construct

5.3.1.1 default(none) clause

The default(none) clause 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 that do not have predetermined data attributes to explicitly appear in a data clause for the compute construct or for a data construct that lexically contains the parallel or kernels construct. To test this clause, the main idea is to initialize the input data arrays on the CPU side and then create them again on the parallel or kernels directive. Therefore these arrays’ original values should not be accessible from the GPU side.
/* Matrix-vector multiplication on CPU only and get known result */
for(int i=0; i<n; i++){
    real_t temp=0.0;
    for(int j=0; j<n; j++){
        temp+=a[i*n+j] * b[j];
    }
    host_sum += temp;
}
printf("host_sum = %lf\n", host_sum);

/* Matrix-vector multiplication on GPU */
#pragma acc kernels default(none) \
create(a[0:n*n], b[0:n]) copy(c[0:n]) copyin(n)
#pragma acc loop gang independent
for(int i=0; i<n; i++){
    real_t temp=0.0;
    #pragma acc loop worker reduction(+:temp)
    for(int j=0; j<n; j++){
        temp+=a[i*n+j] * b[j];
    }
    c[i] = temp;
}

/* add up vector C to device_sum */
for(int i=0; i<n; i++) {
    device_sum += c[i];
}
printf("device sum = %lf\n", device_sum);

if(fabs(host_sum - device_sum) < PRECISION)
    err = 1;
else
    err = 0;

Listing 5.3: Test case for kernels default(none)
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 Listing 5.3. The worker-level loop performs a reduction, after which we check for the summation values both on CPU and GPU side.

### 5.3.1.2 loop vector clause

In an accelerator parallel or kernels region, the vector clause specifies that the iterations of the associated loop or loops are to be executed with vector or SIMD processing. OpenACC 2.0 add the new feature–loop vector[length:]int-expr to specify the vector length on the parallel/kernels loop vector level. If an argument is specified, the iterations will be processed in vector strips of that length; if no argument is specified, compiler should choose an appropriate vector length.

Listing 5.4 shows the test case for this clause. Within a parallel region, the first two nested loop updates the array A with a static value and the second one calculates the summation. Both inner loops are distributed to vector as a length of CORNER\.vector\_length. CORNER tells the specific parser of this case to iterate values (from -1 to 32) for the vector length. Finally we check the precalculated host\_sum and vector\_sum from GPU side.
vector_sum = 0.0;

#pragma acc parallel copyin(A[0:n*n]) reduction(+:vector_sum)
{
    #pragma acc loop gang
    for (int i=0; i<LOOPCOUNT; i++) {
        #pragma acc loop vector
        for (int j=0; j<LOOPCOUNT; j++) {
            A[i*n + j] = STATIC_NUMBER;
        }
    }

    #pragma acc loop gang
    for (int i=0; i<LOOPCOUNT; i++)
        #pragma acc loop vector (length:CORNER_vector_length)
        for (int j=0; j<LOOPCOUNT; j++) {
            vector_sum += A[i*n + j];
        }
}

if (vector_sum != host_sum)
    err=1;

Listing 5.4: Test case for parallel loop vector
5.3.2 Enter Data and Exit Data Directives

5.3.2.1 enter data pcopyin clause

```c
int err_1 = 0, err_2 = 0;

// case 1: pcopyin should be copyin
#pragma acc enter data pcopyin(a[0:n*n], b[0:n]) create(c[0:n])
#pragma acc kernels
#pragma acc loop gang independent reduction(+:device_sum)
for(int i=0; i<n; i++){
  real_t temp=0.0;
  #pragma acc loop reduction(+:temp)
  for(int j=0; j<n; j++){
    temp+=a[i*n+j] * b[j];
  }
  c[i] = temp;
  device_sum += c[i];
}
if (fabs(host_sum - device_sum) >= PRECISION)
  err_1++;

// case 2: pcopyin should be present
#pragma acc enter data pcopyin(c[0:n])
#pragma acc kernels pcreate(c[0:n])
#pragma acc loop gang reduction(+:device_sum)
for(int i=0; i<n; i++){
  device_sum += c[i];
}
#pragma acc exit data delete(a[0:n*n], b[0:n], c[0:n])
if (fabs(device_sum - 2*host_sum) >= PRECISION)
  err_2++;
err = err_1 + err_2;
```

Listing 5.5: Test case for enter data pcopyin
An `enter data` directive may be used to define scalars, arrays and subarrays to be allocated in the device memory for the remaining duration of the program, or until an `exit data` directive that deallocates the data. `pcopyin` means `present_or_copyin`. If the data is already present on the accelerator (GPU), the program behaves as with the `present` clause. No new device memory will be allocated and no data will be moved to or from the device memory. If the data is not present, the program behaves as with the `copy` clause. The data is allocated and copied to the device memory upon `enter` to the region, and copied back to the local memory and deallocated upon `exit` from the region. Therefore, we can first use `copyin` feature to copy the data into GPU memory and the data in the second computation part should be `present` before `exit`. Listing 5.5 gives the implementation of this test. Note that the `pcreate` clause in line 20 will allocate memory for the array `c` (fill with random values or simply 0) if it is not present as expected.

### 5.3.2.2 exit data delete clause

The `delete` clause is used on exit data directives to deallocate arrays, subarrays or common blocks without copying values back to local memory. Listing 5.6 shows the simplified test case for this clause. We initialized the array `c` to all zeros on the host CPU but also transferred to GPU and updated within the “enter/exit data” lifetime. If the array `c` is successfully deleted (deallocated) in the GPU memory, the following kernels `pcopyin` clause will detect it as `copyin` feature. Therefore, the host zero-filled array `c` is “copied” into GPU memory again and the summation result should be nothing but 0.
Listing 5.6: Test case for exit data delete

```cpp
// host c is initialized to 0
#pragma acc enter data copyin(a[0:n*n], b[0:n], c[0:n])

// first kernel, update c
...

#pragma acc exit data delete(a[0:n*n], b[0:n], c[0:n])
// above pragma, c should be deleted from GPU memory

#pragma acc kernels pcopyin(c[0:n])
#pragma acc loop gang reduction(+:device_sum)
for(int i=0; i<n; i++){
    device_sum += c[i];
}
if(device_sum >= 0)
    err = 1;
```

5.3.3 Device Type Clause

One of the main updates for OpenACC 2.0 is the support for multiple device types, along with the current HPC trend that accelerators can be General-Purpose Graphic Processing Units (GPGPUs), Accelerated Processing Units (APUs, [1]), Many Integrated Cores (MIC, [7]), and even ARM-based architectures. OpenACC directives can specify different clauses or clause arguments for different accelerators using the `device_type` clause. Clauses on a directive with no `device_type` clause apply to all accelerator device types. Clauses that follow a `device_type` clause up to the end of the directive or up to the next `device_type` clause are associated with this `device_type` clause.
Listing 5.7: Test case for kernels device_type

Listing 5.7 is a possible method to test the device_type. There are two data regions in this case, one is for NVIDIA GPU and the other for AMD APU. If available,
two accelerators should only execute the kernel assigned to them. The user can also put these two regions into independent test functions or subroutines and launch them in a parallel way.

5.3.4 Routine Bind Clause

As we discussed in Section 4.4.4, the routine directive is used to tell the compiler to compile a given procedure for an accelerator as well as the host. In a file or routine with a procedure call, the routine directive tells the implementation the attributes of the procedure when called on the accelerator. Listing 4.5 is an example of using routine worker clause and routine gang/vector/seq follows the same manner.

The bind clause specifies the name to use when compiling or calling the procedure. If the name is specified as an identifier (bind (name)), it is compiled or called as if that name is specified in the language that is being compiled. If the name is specified as a string (bind (string)), the string is used for the procedure name unmodified. Listing 5.8 is the code layout for both name and string tests. We declare three acc routine: matvec, matvec_another and matvec_host in the program and the first two are defined in another file. Through binding them together, calling procedures in the main driver will forward to their binded target routines: matvec to matvec_another and matvec_alias to matvec_host.
#pragma acc routine(matvec) gang
extern void matvec (real_t* a, real_t* b, real_t* c, int n);

#pragma acc routine(matvec_another) gang
extern void matvec_another (real_t* a, real_t* b, real_t* c, int n);

// call matvec -> matvec_another
#pragma acc routine(matvec_another) bind(matvec)

#pragma acc routine gang bind("matvec_alias")
void matvec_host (real_t* a, real_t* b, real_t* c, int n) {
#pragma acc acc loop gang
    for(int i=0; i<n; i++) {
        real_t temp = 0.0;
        #pragma acc acc loop reduction(+:temp)
        for(int j=0; j<n; j++) {
            temp += a[i*n+j] * b[j];
        }
        c[i] = STATIC_NUMBER;
    }
}

 {...}

// calling function in main driver
#pragma acc acc data copyin(a[0:n*n], b[0:n]) copy(name_c[0:n], string_c[0:n])
{
#pragma acc acc parallel num_gangs(ARRAYSIZE/10)
    {
        matvec (a, b, name_c, n); // i.e. matvec_another
        matvec_alias (a, b, string_c, n); // i.e. matvec_host
    }
}

Listing 5.8: Test case for routine bind
5.3.5 Special Async Arguments

OpenACC 2.0 introduces two special async values: acc_async_noval and acc_async_sync. An async clause with the async-argument acc_async_noval will behave the same as if the async clause had no argument. An async clause with the async-argument acc_async_sync will behave the same as if no async clause appeared.

To divide the problem into the smallest units, we consider all the conditions of using async clause: on the enter, exit, wait, kernels, parallel and update directives. Therefore we have six tests and each of them will try two special async values. The basic idea is that if all outstanding and asynchronous operations have completed, the runtime routine acc_async_test_all will return with a nonzero value when use directives or clauses:

- **wait**: acc_async_noval and acc_async_sync should be equal to wait,

or the runtime routine will act differently under the circumstances:

- **enter/exit data**: return with nonzero on acc_async_noval and zero on the other;

- **parallel/kernels**: between two parallel/kernels regions, return with nonzero on acc_async_sync and zero on the other; *vice versa* for the condition after the two regions;

- **update**: return with nonzero on acc_async_noval and zero on the other.
5.3.6 Atomic Directive

An atomic construct ensures that a specific storage location is accessed and/or updated atomically, preventing simultaneous reading and writing by gangs, workers and vector threads that can result in indeterminate values.

The atomic constructs have four clauses: read, write, update, and capture. An expression-stmt or a structured-block should follow these clauses, which contains many other forms. For example the structured-block is a structured block with one of the following forms:

\[
\begin{align*}
&\{ v = x; x \text{ binop} = \text{expr}; \} \\
&\{ x \text{ binop} = \text{expr}; v = x; \} \\
&\{ v = x; x = x \text{ binop expr}; \} \\
&\{ v = x; x = \text{expr binop } x; \} \\
&\{ x = x \text{ binop expr}; v = x; \} \\
&\{ x = \text{expr binop } x; v = x; \} \\
&\{ v = x; x = \text{expr}; \} \\
&\{ v = x; x++; \} \\
&\{ v = x; ++x; \} \\
&\{ ++x; v = x; \} \\
&\{ x++; v = x; \}
\end{align*}
\]

Moreover, binop is one of +, *, −, /, &, ^, |, <<, or >>. To minimize the testing
feature, we separate four clauses first and then each form needs to be tested independently. A parser script will replace the tag `OP_TARGET` with all the binary operators once a time and generate the test module code. Listing 5.9 shows one of the forms for atomic directive. Both CPU and GPU execute the same operations and compare the results when the computation is complete.

```c
/* 2) {v = x; x = x binop expr;} */

x = STATIC_NUMBER_1;
v = STATIC_NUMBER_2;
#pragma acc parallel loop num_gangs(bin_limits)
for (i=0; i<bin_limits; i++) {
    #pragma acc atomic capture
    {v = x; x = x OP_TARGET expr;}
}

/* host verify */
host_x = STATIC_NUMBER_1;
host_v = STATIC_NUMBER_2;
for (i=0; i<bin_limits; i++) {
    host_v = host_x;
    host_x = host_x OP_TARGET expr;
}
err[2] = (host_x!=x || host_v!=v);
```

Listing 5.9: Test case for atomic directive
5.3.7 Runtime Libraries

5.3.7.1 acc_pcreate

The acc_present_or_create (or acc_pcreate) routine tests to see if the data is already present on the device; if not, it allocates memory on the accelerator device to correspond to the specified host memory, on a non-shared memory device. It is equivalent to the enter data directive with a present_or_create clause.

Similar to the enter_data_pcopyin clause we discussed in Section 5.3.2.1, validating acc_pcreate routine needs two test steps: “create” feature and “present” feature. Following case example (Listing 5.10) uses acc_present_or_create to “create” (allocate) memory for the array b on the device while the array a is copied in via the enter data directive. If b is not created successfully, pcopyin clause will take responsibility to transfer the data from the host memory, which will result in the difference between the summation of the updated array c and precalculated host_sum. Since the next computational kernel is before exit data, the attribute for b should still be alive. Therefore, the second acc_present_or_create needs to take b as present (still in the GPU memory) and accordingly, pcreate clause on the kernels construct makes the same choice. Finally, the computation result remains the same as the one of previous kernels.
// host_sum is pre-calculated

// case 1: b is not present, should create
acc_present_or_create(b, n*sizeof(real_t));

#pragma acc enter data copyin(a[0:n*n]) pcopyin(b[0:n])
#pragma acc kernels pcopyin(a[0:n*n], b[0:n])
#pragma acc loop gang reduction(+:device_sum_create)
for (int i=0; i<n; i++) {
    real_t temp=0.0;
    #pragma acc loop reduction(+:temp)
    for (int j=0; j<n; j++) {
        temp += a[i*n+j] * b[j];
    }
    device_sum_create += temp;
}

// should be different
if (fabs(host_sum - device_sum_create) < PRECISION) sum_err++;

// case 2: still in the lifetime, b should be present now
acc_present_or_create(b, n*sizeof(real_t));

#pragma acc kernels pcopyin(a[0:n*n]) pcreate(b[0:n])
#pragma acc loop gang reduction(+:device_sum_present)
for (int i=0; i<n; i++) {
    real_t temp=0.0;
    #pragma acc loop reduction(+:temp)
    for (int j=0; j<n; j++) {
        temp += a[i*n+j] * b[j];
    }
    device_sum_present += temp;
}
#pragma acc exit data delete(a[0:n*n])

// should be same as the previous device sum
if (fabs(device_sum_create - device_sum_present) > PRECISION) sum_err++;
5.3.7.2 acc_map_data

The acc_map_data routine maps previously allocated device data to the specified host data. It is similar to an enter data directive with a create clause, except instead of allocating new device memory to start a data lifetime, the device address to use for the data lifetime is specified as an argument. Listing 5.11 is a simple example of the acc_map_data routine.

```c
// array host_c is not initialized
#pragma acc enter data copyin(a[0:n*n], b[0:n], c[0:n])
#pragma acc kernels pcopyin(a[0:n*n], b[0:n], c[0:n])
#pragma acc loop
for(int i=0; i<n; i++) {
    real_t temp=0.0;
    #pragma acc loop reduction(+:temp)
    for(int j=0; j<n; j++){
        temp += a[i*n+j] * b[j];
    }
    c[i] = temp;
}
acc_map_data( host_c, acc_deviceptr(c), n*sizeof(real_t) );
acc_update_self(host_c, n*sizeof(real_t));
#pragma acc exit data delete(a[0:n*n], b[0:n], c[0:n])
```

Listing 5.11: Test case for acc_map_data

This case depends on several other runtime routines. After the on-device computation for the array c, acc_deviceptr(c) gets the address for the array c on
the device and `acc_map_data` map it to the host array `host_c`. The subsequent `acc_update_self` updates the host array by transferring the values from GPU side. The test will succeed if we get the same values for `host_c` and `c`. 
Chapter 6

Evaluation Results and Discussion

We use the OpenACC 2.0 validation suite to evaluate the implementation correctness in vendor compilers including PGI, Cray and CAPS Enterprise, both for C and FORTRAN. The entire evaluation process is along with the development of validation suite. We work very closely with three main vendors that offer compiler support for OpenACC and assist them in identifying and resolving compiler bugs helping them improve the quality of their compilers. In return, they provide us with feedback on the design of test cases and clarify the ambiguity for the usage of certain new features. This chapter is written in a manner that we do not simply report about the results and bugs but we also share our experiences in developing and improving the test cases.
6.1 Compilers and Testbeds

During the developing cycle of the OpenACC 2.0 testsuite, compilers from three vendors are upgrading at the same time. The ranges of their compiler versions are:

- **PGI**: 14.1—14.10
- **Cray**: 8.2.2—8.4.0
- **CAPS**: 3.4.0—3.4.5

We test all the stable versions and collect the pass rate for comparison. At the time of compiling the results for the final report, we are informed that CAPS Enterprise will not be in business any more. However, we still validate their OpenACC 2.0 implementations and include the results in this thesis.

For the PGI and CAPS compiler, our test bed is a heterogeneous system consisting of 16 cores Intel Xeon x86 64 CPU, and an NVIDIA Kepler GPU card (K20) with 2496 Stream Processors (SP) and 5GB global memory. The Cray compiler is the XE6 [3] system with Interlagos x86 64 CPU and a more advanced NVIDIA K20X GPU (2688 SPs and 6GB global memory).

6.2 Quantitative Report

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.

The notations used in the following tables are:

- **C**: C language
- **FOR**: FORTRAN language
- **FT**: Functional Test
- **CT**: Cross Test (supposed to fail)
- **P**: Pass
- **F**: Fail
- **CE**: Compilation Error
- **EE**: Execution Error
- **NT**: No Test if not Pass
- **NA**: Not Applicable
For the first round of experiments, we disable the optimization flags to avoid any potential uncertainties, i.e., the code reconstruction while compiling the unit tests. And then we turn on the -O3 optimization as most of the time compiler optimizations are highly used by programmers. We did not find any differences with the turning off/on of the optimizations flags. We tabulate the results for atomic clauses separately since it contains too many forms.

Table 6.1–6.4 shows the experimental results of evaluating the directives on three compilers of their latest versions, which contain Functional and Cross tests using int or double type, both for C and FORTRAN codes. For instance, the kernels_wait is to test the wait clause on the kernels construct; all the names starting with acc_ are testing newly added runtime library routines. Most cross tests are from simply removing the target directives or clauses for functional tests, however, there are exceptions like acc_is_present that needs to change the routine argument.

Table 6.5–6.6 specifically presents the results for atomic C and FORTRAN directive, respectively. We choose not to include double type tests since most atomic operations are more scalar-oriented, which aims on int and bool types (in FORTRAN are integer and logical). Moreover, half of the operations are Bitwise, which is also not applicable for double type data.

Table 6.7 lists the results for the available corner tests [22] of certain features. Corner test mainly targets on the integer parameters and aims to find the edge cases (minimum or maximum value) that may throw out exceptional failures. To design the corner test, we set a range and a test step for integer parameters. For example, [−4, 6, 2] means the range is from −4 to 6 with a step of 2, then the value −4, −2, 0, 2
Table 6.1: Experimental results on three compilers, part 1

<table>
<thead>
<tr>
<th>directives</th>
<th>PGI</th>
<th>Cray</th>
<th>CAPS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FT</td>
<td>CT</td>
<td>FT</td>
</tr>
<tr>
<td>kernels_wait</td>
<td>int</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td>kernels_device_type</td>
<td>int</td>
<td>CE NT</td>
<td>CE NT</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE NT</td>
<td>CE NT</td>
</tr>
<tr>
<td>kernels_default_none</td>
<td>int</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td>parallel_wait</td>
<td>int</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td>parallel_device_type</td>
<td>int</td>
<td>CE NT</td>
<td>CE NT</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE NT</td>
<td>CE NT</td>
</tr>
<tr>
<td>parallel_default_none</td>
<td>int</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td>parallel_implicit</td>
<td>int</td>
<td>P NA</td>
<td>P NA</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P NA</td>
<td>P NA</td>
</tr>
<tr>
<td>enter_data_if_true</td>
<td>int</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td>enter_data_if_false</td>
<td>int</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td>enter_data_async</td>
<td>int</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td>enter_data_wait</td>
<td>int</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td>enter_data_copyin</td>
<td>int</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td>enter_data_create</td>
<td>int</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td>exit_data_if_create</td>
<td>int</td>
<td>F NT</td>
<td>F NT</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>F NT</td>
<td>F NT</td>
</tr>
<tr>
<td>exit_data_if_true</td>
<td>int</td>
<td>F NT</td>
<td>F NT</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>F NT</td>
<td>F NT</td>
</tr>
<tr>
<td>exit_data_if_false</td>
<td>int</td>
<td>F NT</td>
<td>F NT</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>F NT</td>
<td>F NT</td>
</tr>
<tr>
<td>exit_data_async</td>
<td>int</td>
<td>CE NT</td>
<td>CE NT</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE NT</td>
<td>CE NT</td>
</tr>
<tr>
<td>exit_data_wait</td>
<td>int</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td>exit_data_copyout</td>
<td>int</td>
<td>F F</td>
<td>F F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>F F</td>
<td>F F</td>
</tr>
<tr>
<td>exit_data_delete</td>
<td>int</td>
<td>P F</td>
<td>P F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P F</td>
<td>P F</td>
</tr>
</tbody>
</table>
Table 6.2: Experimental results on three compilers, part 2

<table>
<thead>
<tr>
<th>directives</th>
<th>PGI</th>
<th>FOR</th>
<th>Cray</th>
<th>FOR</th>
<th>CAPS</th>
<th>FOR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>FT</td>
<td>CT</td>
<td>FT</td>
<td>CT</td>
<td>FT</td>
<td>CT</td>
</tr>
<tr>
<td>kernels_loop_gang</td>
<td>int</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>P</td>
</tr>
<tr>
<td>kernels_loop_worker</td>
<td>int</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>P</td>
</tr>
<tr>
<td>kernels_loop_vector</td>
<td>int</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>P</td>
</tr>
<tr>
<td>parallel_loop_gang</td>
<td>int</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>P</td>
</tr>
<tr>
<td>parallel_loop_vector</td>
<td>int</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>P</td>
</tr>
<tr>
<td>loop_auto</td>
<td>int</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>P</td>
</tr>
<tr>
<td>loop_tile</td>
<td>int</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>P</td>
</tr>
<tr>
<td>loop_device_type</td>
<td>int</td>
<td>CE</td>
<td>NT</td>
<td>CE</td>
<td>NT</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NT</td>
<td>CE</td>
<td>NT</td>
<td>P</td>
</tr>
<tr>
<td>declare_link</td>
<td>int</td>
<td>CE</td>
<td>NT</td>
<td>CE</td>
<td>NT</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NT</td>
<td>CE</td>
<td>NT</td>
<td>P</td>
</tr>
<tr>
<td>update_self</td>
<td>int</td>
<td>CE</td>
<td>NT</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NT</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>update_wait</td>
<td>int</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>update_device_type</td>
<td>int</td>
<td>CE</td>
<td>NT</td>
<td>CE</td>
<td>NT</td>
<td>CE</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NT</td>
<td>CE</td>
<td>NT</td>
<td>CE</td>
</tr>
<tr>
<td>wait_async</td>
<td>int</td>
<td>CE</td>
<td>NT</td>
<td>CE</td>
<td>NT</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NT</td>
<td>CE</td>
<td>NT</td>
<td>P</td>
</tr>
<tr>
<td>routine_with_name</td>
<td>int</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>routine_gang</td>
<td>int</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>routine_worker</td>
<td>int</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>routine_vector</td>
<td>int</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>routine_seq</td>
<td>int</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>routine_bind_name</td>
<td>int</td>
<td>F</td>
<td>NT</td>
<td>CE</td>
<td>NT</td>
<td>CE</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>F</td>
<td>NT</td>
<td>CE</td>
<td>NT</td>
<td>CE</td>
</tr>
<tr>
<td>routine_bind</td>
<td>int</td>
<td>CE</td>
<td>NT</td>
<td>CE</td>
<td>NT</td>
<td>EE</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NT</td>
<td>CE</td>
<td>NT</td>
<td>EE</td>
</tr>
<tr>
<td>routine_device_type</td>
<td>int</td>
<td>CE</td>
<td>NT</td>
<td>CE</td>
<td>NT</td>
<td>CE</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NT</td>
<td>CE</td>
<td>NT</td>
<td>P</td>
</tr>
</tbody>
</table>
Table 6.3: Experimental results on three compilers, part 3

<table>
<thead>
<tr>
<th>directives</th>
<th>PGI</th>
<th>Cray</th>
<th>CAPS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>FOR</td>
<td>C</td>
</tr>
<tr>
<td>routine_nohost</td>
<td></td>
<td>FT</td>
<td>FT</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>noval_enter_data</td>
<td></td>
<td>F</td>
<td>NT</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>noval_exit_data</td>
<td></td>
<td>F</td>
<td>NT</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>noval_parallel</td>
<td></td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>noval_kernels</td>
<td></td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>noval_update</td>
<td></td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>noval_wait</td>
<td></td>
<td>F</td>
<td>NT</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>sync_enter_data</td>
<td></td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>sync_parallel</td>
<td></td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>sync_kernels</td>
<td></td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>sync_update</td>
<td></td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>sync_wait</td>
<td></td>
<td>F</td>
<td>NT</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>acc_wait_enter</td>
<td></td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>acc_wait_exit</td>
<td></td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>acc_wait_wait</td>
<td></td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>acc_wait_kernels</td>
<td></td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>acc_wait_parallel</td>
<td></td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>acc_wait_update</td>
<td></td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>acc_wait_sharing</td>
<td></td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>acc_wait_all_kernels</td>
<td></td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
</tr>
</tbody>
</table>
Table 6.4: Experimental results on three compilers, part 4

<table>
<thead>
<tr>
<th>directives</th>
<th>PGI</th>
<th>FOR</th>
<th>Cray</th>
<th>FOR</th>
<th>CAPS</th>
<th>FOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc_wait_all_parallel</td>
<td>int</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>acc_wait_all_sharing</td>
<td>int</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>acc_copyin</td>
<td>int</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>acc_present_or_copyin</td>
<td>int</td>
<td>F</td>
<td>NT</td>
<td>P</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>F</td>
<td>NT</td>
<td>P</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>acc_create</td>
<td>int</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>acc_present_or_create</td>
<td>int</td>
<td>P</td>
<td>F</td>
<td>NT</td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>F</td>
<td>NT</td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>acc_copyout</td>
<td>int</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>acc_update_device</td>
<td>int</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>acc_update_self</td>
<td>int</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>F</td>
<td>P</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>acc_map_data</td>
<td>int</td>
<td>P</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>acc_unmap_data</td>
<td>int</td>
<td>EE</td>
<td>NT</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>EE</td>
<td>NT</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>acc_deviceptr</td>
<td>int</td>
<td>EE</td>
<td>NT</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>EE</td>
<td>NT</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>acc_hostptr</td>
<td>int</td>
<td>F</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>acc_is_present</td>
<td>int</td>
<td>P</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>acc_memcpy_to_device</td>
<td>int</td>
<td>P</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>acc_memcpy_from_device</td>
<td>int</td>
<td>P</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>nest_kernels_kernels</td>
<td>int</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>nest_parallel_kernels</td>
<td>int</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>nest_kernels_parallel</td>
<td>int</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>nest_parallel_parallel</td>
<td>int</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NA</td>
<td>CE</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>multiarray</td>
<td>int</td>
<td>P</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>P</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>data_pointer</td>
<td>int</td>
<td>CE</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>CE</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
Table 6.5: Experimental results of C atomic clauses

<table>
<thead>
<tr>
<th>Forms</th>
<th>PGI</th>
<th>Cray</th>
<th>CAPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_read</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>atomic_write</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>update or not present</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x++;</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>x--;</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>++x;</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>--x;</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>x binop= expr;</td>
<td>CE</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>x = x binop expr;</td>
<td>CE</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>x = expr binop x;</td>
<td>CE</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>capture</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x++;</td>
<td>P</td>
<td>CE</td>
<td>P</td>
</tr>
<tr>
<td>x--;</td>
<td>P</td>
<td>CE</td>
<td>P</td>
</tr>
<tr>
<td>++x;</td>
<td>P</td>
<td>CE</td>
<td>P</td>
</tr>
<tr>
<td>--x;</td>
<td>P</td>
<td>CE</td>
<td>P</td>
</tr>
<tr>
<td>x binop= expr;</td>
<td>CE</td>
<td>CE</td>
<td>P</td>
</tr>
<tr>
<td>x = x binop expr;</td>
<td>CE</td>
<td>CE</td>
<td>P</td>
</tr>
<tr>
<td>x = expr binop x;</td>
<td>CE</td>
<td>CE</td>
<td>P</td>
</tr>
<tr>
<td>{x = expr binop x; v = x;}</td>
<td>CE</td>
<td>CE</td>
<td>P</td>
</tr>
<tr>
<td>capture: structured block</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>{v = x; x binop= expr;}</td>
<td>CE</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>{x binop= expr; v = x;}</td>
<td>CE</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>{v = x; x = x binop expr;}</td>
<td>CE</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>{v = x; x = expr binop x;}</td>
<td>CE</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>{x = x binop expr; v = x;}</td>
<td>CE</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>{x = expr binop x; v = x;}</td>
<td>CE</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>{v = x; x = expr;}</td>
<td>CE</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>{v = x; x++;}</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>{v = x; ++x;}</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>{++x; v = x;}</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>{x++; v = x;}</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>{v = x; x--;}</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>{v = x; --x;}</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>{--x; v = x;}</td>
<td>P</td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>{x--; v = x;}</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
</tbody>
</table>
Table 6.6: Experimental results of FORTRAN atomic clauses

<table>
<thead>
<tr>
<th>Operators</th>
<th>PGI</th>
<th>Cray</th>
<th>CAPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_read</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>atomic_write</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>update or not present</td>
<td></td>
<td>x = x operator expr</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>-</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>*</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>/</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>.and.</td>
<td>CE</td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>.or.</td>
<td>CE</td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>.eqv.</td>
<td>CE</td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>.neqv.</td>
<td>CE</td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>update or not present</td>
<td></td>
<td>x = intrinsic_procedure_name (x, expr-list)</td>
<td></td>
</tr>
<tr>
<td>max</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>min</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>iand</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>ior</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>ieor</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>capture update-statement; capture-statement</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>-</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>*</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>/</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>.and.</td>
<td>CE</td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>.or.</td>
<td>CE</td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>.eqv.</td>
<td>CE</td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>.neqv.</td>
<td>CE</td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>max</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>min</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>iand</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>ior</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>ieor</td>
<td>P</td>
<td>P</td>
<td>P</td>
</tr>
</tbody>
</table>
will be tested. We are expecting that the compiler should only apply specific range of values due to the total number of threads that a GPU can launch or the loop size.

Table 6.7: Available Corner Test Results

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Clause</th>
<th>Space</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAPS</td>
<td>tile(size)</td>
<td>[MIN_VAL, MAX_VAL, 1]</td>
<td>No Error; when size is greater than the loop size or negative value, tiling size being applied becomes randomly</td>
</tr>
<tr>
<td>Cray</td>
<td>gang(num)</td>
<td>[MIN_VAL, MAX_VAL, 1]</td>
<td>No Error; compiler will always apply a more appropriate num</td>
</tr>
<tr>
<td></td>
<td>worker(num)</td>
<td>[MIN_VAL, MAX_VAL, 1]</td>
<td>Warning; compiler will always apply a more appropriate num</td>
</tr>
<tr>
<td></td>
<td>vector(length)</td>
<td>[MIN_VAL, MAX_VAL, 1]</td>
<td>Warning; compiler will always apply a more appropriate length</td>
</tr>
</tbody>
</table>
6.3 Results Analysis

Figures 6.1–6.3, shows plots of both C and FORTRAN OpenACC compilers along with number of bugs discovered in several stable (even minor number [54]) compiler versions. We tabulate the results in the Table 6.8 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. Note that, the results are for Functional tests of double data type, the others follow the same pattern.

Figure 6.1 shows the bugs discovered with PGI’s OpenACC compiler and how they are rectified over a period of time. PGI begins to provide support for OpenACC 2.0 from version 14.1 onwards. PGI 14.2 shows a much lower pass rate since the atomic directive had not passed the test scenarios [13]. Probably due to priority given to other important directives such as enter/exit data, routine, parallel and kernels. Moreover, one can simply use data directives instead of enter/exit data directives. The pass rate shows a substantial improvement from version 14.4 onwards since it adds the partial support for atomic directive. Later most of the tests that do not pass are mainly due to the async and wait clause before version 14.7, which we will discuss in the next section.

Figure 6.2 shows the plots of how the Cray compiler evolved over a period of time. We see that the pass rates for Cray 8.2.x are lower than >8.3.x versions. However after 8.3.x, the pass rate changes slightly and some bugs still remain unsolved (especially for the FORTRAN tests).
Figure 6.1: Test pass rate for PGI compiler and their versions

Figure 6.2: Test pass rate for Cray compiler and their versions
Figure 6.3 shows the plots for CAPS compiler. The bar plots mostly shows no huge variation on >3.4.1 versions. CAPS 3.4.1 has a lower pass rate due to the failures on the new arguments of `async` clause and `routine` directive. We discuss other probable reasons in the next section.

![Figure 6.3: Test pass rate for CAPS compiler and their versions](image)

### 6.4 Bugs-Identified Examples

In this section, we discuss some of the qualitative and quantitative collection of results about the bugs that we found in the OpenACC compilers for the 2.0 specification. Typically, a bug can be due to a number of reasons: nonconformance to the specification, compilation, runtime, or validation errors.
Table 6.8: Bugs identified in different compilers

<table>
<thead>
<tr>
<th>Compiler</th>
<th>PGI</th>
<th>Cray</th>
<th>CAPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>14.2</td>
<td>14.4</td>
<td>14.6</td>
</tr>
<tr>
<td>Language</td>
<td>C</td>
<td>FOR</td>
<td>C</td>
</tr>
<tr>
<td>Bugs</td>
<td>76</td>
<td>64</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>39</td>
<td>30</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td>Version</td>
<td>14.9</td>
<td>14.10</td>
<td></td>
</tr>
<tr>
<td>Language</td>
<td>C</td>
<td>FOR</td>
<td>C</td>
</tr>
<tr>
<td>Bugs</td>
<td>39</td>
<td>39</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Version</td>
<td>8.2.2</td>
<td>8.2.4</td>
<td>8.2.6</td>
</tr>
<tr>
<td>Language</td>
<td>C</td>
<td>FOR</td>
<td>C</td>
</tr>
<tr>
<td>Bugs</td>
<td>19</td>
<td>25</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>19</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>21</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>14</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>Version</td>
<td>8.3.2</td>
<td>8.4.0</td>
<td></td>
</tr>
<tr>
<td>Language</td>
<td>C</td>
<td>FOR</td>
<td>C</td>
</tr>
<tr>
<td>Bugs</td>
<td>18</td>
<td>21</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>12</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Version</td>
<td>3.4.1</td>
<td>3.4.2</td>
<td>3.4.3</td>
</tr>
<tr>
<td>Language</td>
<td>C</td>
<td>FOR</td>
<td>C</td>
</tr>
<tr>
<td>Bugs</td>
<td>21</td>
<td>37</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>18</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>28</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>19</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>18</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Listing 6.1: Test case for \texttt{exit data if}

```c
// host_sum is pre-calculated on CPU

#pragma acc exit data if(flag) copyout(c[0:n]) delete(a[0:n*n], b[0:n])

// if(1) should move out array c

for (int i=0; i<n; i++) {
    temp_c[i] = c[i];
}

device_sum = 0.0;

for (int i=0; i<n; i++) {
    device_sum += temp_c[i];
}
```

**PGI—\texttt{exit data if(TRUE)}**: OpenACC 2.0 adds \texttt{enter/exit data} directives to control the lifetime of input or output data. If used on the \texttt{exit data} directive, clause \texttt{if} will conditionally deallocate device memory and move data to the device. Listing 6.1 shows a simplified test case for the \texttt{TRUE} condition.
After calculating the array \(c\) on the device, the data should be moved out to
the host array \(c\) since the data movement is enabled by \textbf{if}(TRUE). However, PGI
compiler returns 0.0 for all the array \(c\) and the summation is 0.0 as well, which is
definitely a runtime error. We also found the similar bug in its FORTRAN compiler
for \textbf{if}(FALSE). This bug is still not fixed in the later versions of compiler releases.

```c
#include <stdio.h>

#define N 10

#pragma acc routine seq
void matvec (real_t* a, real_t* b, real_t* c, int N, int i) {
    real_t temp=0.0;
    #pragma acc loop auto reduction(+:temp)
    for(int j=0; j<N; j++) {
        temp+=a[i*N+j] * b[j];
    }
    c[i] = temp;
}

// calling region
#pragma acc data copyin(a[0:n*n], b[0:n], idx) copy(c[0:n])
{
    #pragma acc parallel loop gang num_gangs(N/10) vector_length(1)
    for (idx=0; idx<N; idx++) {
        matvec(a, b, c, N, idx);
    }
}
```

Listing 6.2: Test case for \texttt{routine seq}

PGI—\textit{routine seq}: The \texttt{seq} clause specifies that the procedure does not contain,
nor does it call another procedure that contains a loop with a \texttt{gang}, \texttt{worker}, or
\texttt{vector} clause. A loop in this procedure with an \texttt{auto} clause will be executed in \texttt{seq}
mode. The basic idea is to check whether the loop with \texttt{auto} clause executes in a
sequential manner. Listing 6.2 is a code snippet of the implementation. It regulates the routine matvec to be executed on the worker level and the thread number should be 1 since we specifically set the vector_length as 1. However, the compilation and execution results indicates that the thread number is dynamically decided by the runtime which is more than 1. We found this bug in the PGI 14.1 and it was fixed from version 14.6.

**PGI**—acc_present_or_create(): As we discussed in the Section 5.3.7.1 and Listing 5.10, there are two steps for this test: create and present. However in the PGI FORTRAN compiler, only the present feature gives the right result. It fails to allocate the device memory for the specified array and the program draws back to use default pcopy data attribute, which incorrectly moves the host data into the device memory. This bug is fixed in the later version.

**Cray, CAPS**—dynamic allocating: For the real-world scientific applications, most memory allocations are during the runtime due to the uncertainty of the data size. Therefore it is better to use dynamic allocating (malloc) instead of static way. However, Cray and CAPS compilers fail several tests using dynamic allocating while their static allocating versions pass successfully. This bug is partially fixed in the latest versions of Cray and CAPS compiler, problem remains in tests such as routine_with_name, routine_gang, declare_link and etc.

**Cray**—routine nohost: The nohost tells the compiler not to compile a version of this procedure for the host. All calls to this procedure must appear within accelerator compute regions. This is a typical false positive [53] case because if we call this routine from the host side (Listing 6.3), ideally, only a compilation error or an
execution error proves the correct compiler implementation for this clause.

```
#pragma acc routine nohost
void matvec (real_t *restrict a, real_t *restrict b, real_t *restrict c, int n) {
    ...
}

// calling matvec on CPU should be inaccessible!
matvec(a, b, host_c, n);
```

Listing 6.3: Test case for routine nohost

However, Cray C/FORTRAN compiler chooses not to distinguish the `matvec` routine for host and device; the array `host_c` still gets the return value. This bug is fixed since Cray 8.3.2 compiler.

**Cray**—`acc_is_present_CT`: Runtime library `acc_is_present` routine tests whether a host variable or an array region is present on the device. This is an interesting case for explaining the unusual cross test (`_CT`) design. Listing 6.4 is the simplified code snippet. For the regular functional test, the uncomment part (Line 2–9) judges the array `c` is not present anymore, which is not correct since `exit data` only deallocates the array `a` and `b` on the device. Therefore, testing “is present” fails. However, the CT version successfully detects the unavailability of the array `b`. In other words, testing “is not present” passes. This bug is fixed in the latest Cray compiler.
Listing 6.4: Test case for `acc_is_present`

**CAPS—`wait_async`:** If there is an `async` clause on the `wait` directive, no new operation may be launched or executed on the `async` device activity queue until all operations enqueued up to this point by this thread on the asynchronous activity queues associated with the wait argument have completed. The main idea of the test (Listing 6.5) is to launch two `async` parallel regions in different streams; let parallel region 2 depends on the results of parallel region 1. Using `wait(1)` on parallel region 2 will force it to wait until parallel region 1 completes, in addition to that, parallel region 2 will get incomplete results. However, program fails to wait region 1 and continues to region 2. This bug is still not fixed in the latest version of compiler release.
Listing 6.5: Test case for wait async

6.5 Other Interesting Observations

This section shares several interesting observations in the OpenACC 2.0 specification we found in the process of developing the testsuite and, most of the issues need clearer explanation in the future specification.
**Nested Parallelism:** This is a truly discussion hotspot. OpenACC 2.0 introduces this feature for multidevices. Under this circumstances, the accelerator may also create and launch parallel kernels like host CPUs, allowing for nested parallelism. Therefore, the OpenACC directives may be executed by a host thread or an accelerator thread. However, the specification does not give the expected models of different nesting scenarios. Section 1.2 presents a good opening example for the ambiguity of nested parallelism in a perspective of software testing.

**Behavior with C/C++ pointer variable:** The specification mentions about this change “the behavior of a data clause with a C or C++ pointer variable has been clarified” without further explanations. By comparing to the regular variable, one possible understanding might be: the pointer as the parameter in the data clauses will be treated as scalar variable. The test results shows that only CAPS compiler team implements this feature under the same idea.

**Loop Tile:** This is an advanced feature introduced by OpenACC 2.0. The tile clause specifies that the implementation should split each loop in the loop nest into two loops, with an outer set of tile loops and an inner set of element loops. It seems to be another way to specify different levels of parallelism on accelerators, for better fitting the loops into gang, worker or vector. Listing 6.6 is our testing idea for 1D and 2D tiling. Unfortunately, based on the current implementation of available compilers, there is no way to verify whether the tile works as expected. In the future releases, compilers should give prompt information about the application of loop tiling either at the compilation time or the runtime.
Listing 6.6: Test case for loop tile

6.6 Interactive Feedback

Like other programming models, implementation variations first come from the different understanding of the specification itself. Hence, the prompt and highly efficient iterations with vendor teams play important roles during the development period of our OpenACC 2.0 validation suite.

In order to make this effort successful, we have been collaborating with vendors,
NVIDIA/PGI, Cray and CAPS since the inception of the new 2.0 specification. We work very closely with the vendor team. We identify and report bugs found in their OpenACC 2.0 implementations. We also maintain separate online documents for tracking the current validation status for all the new features. The vendors fix the bugs and inform us when a newer version of the compiler is released. We then verify if the issues are resolved and update the tracking documents. We receive a feedback not only from the vendors but also from the users (mainly from our HPCTools Group). The benefit here is two-fold. The feedback helps us to improve the quality of the test cases and at the same time validate the more thoroughly conformance of the features to the specification.
Chapter 7

Conclusion and Future Work

7.1 Conclusion

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 the conformance to the specification. 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.

We cover the entire new feature set of the OpenACC 2.0 specification. Our
testsuite consists of over 120 regular test cases (both C and FORTRAN). We maintain an online document to capture the “Pass” or “Fail” against each feature implemented by the three compilers. 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 as open bug by the application developers from each vendor team.

It is quite challenging to analyze the reason behind why a compiler will fail certain tests. But our validation suite still tries to provide as much detail as necessary to the compiler developers in order to assist them in improving the implementation of the features in the compiler. Also we believe that the validation suite will help resolve ambiguities in the OpenACC specification and help refine the same if necessary. Finally, considering the clear structure and detailed implementations of OpenACC 2.0 testsuite, it is safe to say that this validation suite can be a very good resources or tutorials for the OpenACC beginners.

7.2 Future Work

As a future work our focus lies in further improving the implementation in the validation suite by firstly, including optimizations on the cross test and the corner test—quite of them still need developers’ experience on designing the appropriate testing strategy, especially for the unusual functional tests. This will contribute
towards reducing the manual interference during the testing process. Secondly, we would like to incorporate more accelerators into our device_type test, which is incomplete in this version of suite due to a lack of computing resources. Thirdly, we would like to widen the coverage of tests by introducing several combinations of the features. However as one can imagine, this cannot be a thoroughly complete task since there may be several different permutations and combinations of features coexisting with one another. A possible way can be porting current benchmarks into OpenACC applications. Last but not least, in another direction, we will also be exploring the OpenACC compiler implementation in a manner of performance evaluation, by timing the compilation and execution, calculating the speedup and GFLOPS for a much larger amount of input data.
Bibliography


