OpenARC: Extensible OpenACC Compiler Framework for Directive-Based Accelerator Programming Study

Seyong Lee
Oak Ridge National Laboratory
lees2@ornl.gov

Jeffrey S. Vetter
Oak Ridge National Laboratory and
Georgia Institute of Technology
vetter@computer.org

***Abstract*—Directive-based, accelerator programming models such as OpenACC have arisen as an alternative solution to program emerging Scalable Heterogeneous Computing (SHC) platforms. However, the increased complexity in the SHC systems incurs several challenges in terms of portability and productivity. This paper presents an open-sourced OpenACC compiler, called OpenARC, which serves as an extensible research framework to address those issues in the directive-based accelerator programming. This paper explains important design strategies and key compiler transformation techniques needed to implement the reference OpenACC compiler. Moreover, this paper demonstrates the efficacy of OpenARC as a research framework for directive-based programming study, by proposing and implementing OpenACC extensions in the OpenARC framework to 1) support hybrid programming of the unified memory and separate memory and 2) exploit architecture-specific features in an abstract manner. Porting thirteen standard OpenACC programs and three extended OpenACC programs to CUDA GPUs shows that OpenARC performs similarly to a commercial OpenACC compiler, while it serves as a high-level research framework.**

***Keywords*—Compilers,Programming Techniques,Programming Environments,Code generation**

I. Introduction

Scalable Heterogeneous Computing (SHC) platforms enabled by heterogeneous devices, such as general purpose graphics processors (NVIDIA CUDA and AMD GCN), Intel Xeon Phi, etc., are emerging as an alternative solution to respond with the hardware constraints in the today's architectures [22], [6], [3]. However, the heterogeneity in SHC puts a significant burden on its programming: to make use of these platforms, scientists must deal with multiple, different programming models (e.g., MPI, OpenMP, CUDA, and OpenCL) simultaneously and apply different optimization strategies, depending on the target architectures and used programming models, incurring portability and productivity issues. Directive-based accelerator programming models [4], [5], [7], [11], [16], [14], [15] are one such attempt to address these challenges. Among them, OpenACC [14] is the first standardization effort to provide programming/performance portability across different device types and compiler vendors. A major benefit of OpenACC (and other directive-based accelerator programming models) is that it provides very high-level abstractions over the complexity of the underlying heterogeneous architectures and low-level programming languages such as CUDA and OpenCL. However, these abstractions should be balanced out to compromise between two opposite goals: programmability and performance. In previous work, we investigated existing directive-based programming models and their implementations in order to better understand this balance on these emerging programming models [10]; we identified several important issues:

Functionality. 1) Most of them either do not provide reduction clauses or support only scalar reductions. 2) Synchronization is supported in limited ways. 3) All work on array-based computations, but pointer operations are supported limitedly. 4) Each model has different restrictions on the types of mappable regions. These different limits on the functionality of each model incur inconsistent performance and portability problems.

Tunability. Most of the existing models do not provide enough control over various architecture-specific features (e.g., specialized memory) and their respective compiler-optimizations as evidenced by their measured performance. This lack of control limits the programming model tunability.

Debuggability. Directive models provide high-level abstraction, but the opaque nature of these models put a significant burden on the user for debugging. They do not provide insight to the users on the multiple levels of translation, or properly attribute performance data to application constructs.

To investigate these issues in the directive-based accelerator programming, we have developed a new research compiler framework, called OpenARC [9], which supports full features of OpenACC standard V1.0 (and subset of V2.0) and offers built-in performance and debugging tools, in addition to various compile-time analysis and transformation tools. Here are the main contributions of this paper:

- This paper provides an overview of the important design strategies and several key transformation techniques needed to realize a reference OpenACC compiler in the OpenARC framework, which includes *global variable propagation* and *OpenACC worker-single mode transformation* techniques, while detailed description of the base implementation of OpenARC is explained in a reference OpenARC paper [9].
- We demonstrate the effectiveness of OpenARC as a general research framework with illustration. Examples include 1) the study on the impact of array reduction on GPU computing, 2) OpenACC extension to support

978-1-4799-7023-0/14 $31.00 © 2014 IEEE
DOI 10.1109/WACCPD.2014.7

IEEE computer society

hybrid programming of both the unified memory and the separate memory, 3) device-aware OpenACC extensions to support architecture-specific features, and 4) directive-based, interactive program debugging and optimization study (More details on this debugging study can be found in [8].).

- We evaluate OpenARC by porting thirteen standard OpenACC programs and three extended OpenACC programs to NVIDIA GPUs and comparing the performance against a commercial OpenACC compiler and hand-written CUDA programs. The results show that OpenARC performs similarly to the commercial compiler, while it serves as a high-level research framework.

The rest of this paper is organized as follows: Sect. II gives an overview of the OpenACC programming model, Sect. III explains the overall design goals and key transformation techniques needed to implement the OpenACC compiler (OpenARC). Applications of the OpenARC framework, evaluation, and conclusions are presented in Sect. IV, Sect. V, and Sect. VI, respectively.

II. Background: OpenACC Accelerator Programming Interface

OpenACC [14] is a directive-based, accelerator programming interface, which is initiated by a consortium of CAPS, CRAY, PGI, and NVIDIA as the first standardization effort to provide portability across device types and compiler vendors. The OpenACC Application Programming Interface (API) consists of the compiler directives, library routines, and environment variables. The OpenACC API allows programmers to provide high-level information, known as "directives", to the compiler, identifying which regions of a host program to be offloaded to an accelerator device (*compute regions*), and how offloaded regions will be executed (*execution modes*), without requiring programmers to modify or adapt the underlying code itself. Then, the underlying OpenACC compiler will handle all the complex details, such as accelerator code generation, data transfers between a host and a device, accelerator initialization, etc., by generating an actual $host + device$ program for the target accelerator.

OpenACC has two major types of directives: directives for managing parallelism and those for managing data. The directives for parallelism guide types of parallelism to execute loops, and those for data deal with data mapping and transfers between the host and the accelerator. OpenACC provides two types of *compute regions*: *kernels region*, which typically contains one or more work-sharing loops and may be divided into a sequence of device kernels by the compiler, and *parallel region*, which may also contain multiple work-sharing loops but will be executed as a single device kernel on the accelerator. OpenACC supports three levels of parallelism: 1) *gang* parallelism is coarse-grain; a number of gangs will be launched on the device to execute a device kernel. 2) *Worker* parallelism is fine-grain; each gang will have one or more workers. 3) *Vector* parallelism represents Single Instruction, Multiple Data (SIMD) or vector operations within a worker. These three work-sharing clauses guide how to execute loops in a compute region on the target device, but actual parallelism mapping onto a device may depend on the device capability and the compiler implementation. For example, the CUDA programming model supports two levels of parallelism; *thread block*-level parallelism, which is coarse-grain, and *thread*-level parallelism, which is fine-grain, and CUDA-supporting GPUs implement thread-level parallelism using an SIMD architecture. Therefore, OpenACC compilers may choose different strategies to map three-level OpenACC parallelism onto CUDA GPUs.

When executing a compute region on the device, one or more gangs are launched, each of which may contain one or more workers. Each worker has the capability to execute at least one vector lane. The compute region starts in *gang-redundant* mode, where one vector lane of one worker in each gang executes the same code redundantly. When the program encounters a gang loop, the execution mode is changed to *gang-partitioned* mode, where the iterations of the gang loop are partitioned across gangs, but still with only one vector lane of one worker in each gang active. If a gang reaches a worker loop, the gang switches from *worker-single* mode, where only one worker is active, to *worker-partitioned* mode, where all workers in a gang are active, and the iterations of the worker loop are partitioned across workers of this gang, but still with one active vector lane per worker (*vector-single* mode). If a worker encounters a vector loop, the worker transitions to *vector-partitioned* mode, where all vector lanes in the worker execute assigned iterations of the vector loop using SIMD or vector operations. If a single loop is annotated with multiple work-sharing clauses, the iterations of the loop will be distributed across gangs, workers, and vector lanes as appropriate.

III. OpenARC: Open Accelerator Research Compiler

OpenARC [9] is the first open-source compiler supporting full features of OpenACC V1.0, which takes C-based, input OpenACC programs and generates architecture-specific output codes. This paper primarily focuses on targeting CUDA GPUs, but OpenARC has recently been extended to support both CUDA and OpenCL back ends. More details on porting to OpenCL devices (e.g., Intel MIC and AMD GCN architectures) using OpenARC are shown in a companion paper [19]. There exist open-source implementations of OpenACC; accULL [18] provides a source-to-source translation framework combined with a general runtime system that can work with OpenACC, but it is developed as a fast-prototyping tool with limited contexts, while OpenARC supports full research contexts with a fine-grained control over the overall translations. OpenUH [21] also provides a source-to-source translation of OpenACC; as a branch of the open source Open64 compiler suite, OpenUH provides a rich set of compiler components, such as multiple frontends supporting various input languages, a multi-level IR, and multi-level optimization passes. However, the complex multi-level IR intertwined with the multi-level optimizations makes it difficult to perform various high-level optimizations for directive-based accelerator programming study. Another open-source accelerator compiler, HOMP [12], is based on the ROSE compiler infrastructure [17], but it targets the OpenMP accelerator model, rather than OpenACC. The differences between HOMP and OpenARC in terms of the target models and design strategies will make them complement each other by providing diverse research environment for accelerator programming.

This section presents the overall design of OpenARC and important compile-time transformations required for efficient, automatic porting of OpenACC programs into a target accelerator. Detailed description of the base OpenARC implementation can be found in the reference OpenARC paper [9].

A. Design Goals

To serve as a general research compiler framework, OpenARC has been designed with the following emphases: extensibility, debuggability, and tunability.

1) Extensibility: OpenARC is designed with extensibility in mind. OpenARC, which is extended from the Cetus compiler infrastructure [2], is equipped with various advanced analysis and transformation techniques. They can be used as base building blocks to easily create more advanced compiler passes. OpenARC's very high-level IR class hierarchy can convey common program semantics in traditional general-purpose languages in a language independent way. To capture new language features not available in traditional languages, OpenARC's IR class hierarchy can be easily extended. As an alternative, OpenARC's rich, extensible annotations can capture various semantic information [9]. This extensible design allows OpenARC to provide a powerful research framework for various programming studies.

2) Debuggability: OpenARC generates output GPU code by unparsing very high-level IR, resulting in output code similar to the input code, which allows more readability than existing OpenACC compilers. Moreover, there is a clear separation between analysis passes and transformation passes, and all compiler passes communicate with each other through annotations, which allows OpenARC to build various traceability and instrumentation mechanisms into the process. This method establishes links between input directive models and output codes/performance so that users can attribute performance measurements appropriately.

3) Tunability: As a directive compiler for OpenACC, OpenARC has rich directives/environment variables for internal tracing and GPU-specific optimizations. Combined with its built-in tuning tools, OpenARC allows users to control overall OpenACC-to-GPU translation and optimization in a fine-grained, but still abstract manner, offering very high tunability.

B. Compiler Transformations

OpenARC exploits various advanced analysis and transformation techniques to efficiently port OpenACC applications to a target accelerator. This section describes several key transformation techniques used to address various issues arising during the OpenACC-to-CUDA translation.

1) Global Variable Propagation: If an OpenACC program is ported to a device that has a separate address space from a host, which is true in most GPUs, all data accessed in a compute region should be allocated on the device memory, all references to the host data in the region should be replaced with those to the corresponding data on the device memory. This mapping requires that all references to the global data in a device function called in a compute region should be explicitly passed to the function as arguments, unless all parent device functions calling the device function and the device function itself are fully inlined in the compute region. However, full inlining may decrease the code readability significantly, worsening debugging problems, and may lose array shape information with additional aliasing, preventing some advanced compiler optimizations. (Parameter passing also incurs pointer aliasing, but they occur only at function boundaries and preserve original shape information, much less complex than those incurred by full inlining.) For these reasons, OpenARC performs a *global variable propagation* transformation (algorithm 1) for the global data accessed by device functions, instead of doing full inlining. The transformation also has a side benefit making it easier for a backend compiler (e.g., CUDA NVCC) to perform inlining, if necessary. The algorithm transforms device functions accessing global variables in depth-first-traversal of a function-call graph, so that global variables accessed in a callee function are passed as function arguments under the caller-function's context. Even though not shown in algorithm 1, it also performs several optimizations, such as passing scalar variables by value if they are read-only.

Algorithm 1 Global Variable Propagation Transformation

Input: input OpenACC C program
Output: OpenACC C program where global variables are passed as function parameters for device functions in compute regions

```
Create an empty F2GPmap, which will contain mapping of [function, [global symbol, function parameter]]
for each device function called in compute regions do
    DEVPROCCLONE(device function, null)
function DEVPROCCLONE(dFunc, callerFunc)
    G2Pmap = F2GPmap.get(dFunc)        ▷ create [dFunc, G2Pmap] mapping and add it to F2GPmap, if not existing
    CallerG2Pmap = F2GPmap.get(callerFunc)    ▷ null if not existing
    cFunc = a cloned function of dFunc (create new one if not existing)
    cFCall = a new function call for cFunc; replace dFunc call site with this
    for each global symbol (gsym) interprocedurally accessed in dFunc do
        if CallerG2Pmap exists then arg = CallerG2Pmap.get(gsym) else arg = gsym
        if cFunc is newly created then
            param = a new function parameter for gsym, added to cFunc
            G2Pmap.put(gsym,param)
        Add arg to cFCall
    for each function (nDFunc) called in dFunc do
        DEVPROCCLONE(nDFunc, dFunc)
end function
```

2) OpenACC Worker-Single Mode Transformation: As explained in Sect. II, the execution mode of a compute region can change according to work-sharing rules. Therefore, enforcing exact execution mode to each program statement is necessary for correct program execution. In porting OpenACC to CUDA, OpenARC maps gangs to CUDA thread blocks and workers to CUDA threads, ignoring vector parallelism. In this mapping strategy, codes in *worker-single* mode should be executed only by one worker in a gang, meaning only one thread in

Algorithm 2 OpenACC Worker-Single Mode Transformation

Input: input OpenACC C program
Output: OpenACC C program where worker-single mode is enforced

1: **for** each compute region without worker clause **do**
2: HANDLEWSMODE(region body, true)
3: **function** HANDLEWSMODE(region, TopRegion)
4: WSMode = true; Groups = an empty list; tGrp = an empty group
5: **for** each statement in region **do**
6: Split = false
7: **if** declaration or control-flow statement **then** Split = true
8: **else if** barrier annotation or loop with worker clause **then**
9: WSMode = false; Split = true
10: **else if** has child regions or function calls **then**
11: **if** HANDLEWSMODE(child region/function, false) is false **then**
12: WSMode = false; Split = true
13: **if** Split is true **then** ▷ split a group.
14: **if** tGrp is not empty **then** Groups.add(tGrp); tGrp = a new group
15: **else** tGrp.add(current statement)
16: **if** tGrp is not empty **then** Groups.add(tGrp)
17: **if** WSMode is false or TopRegion is true **then**
18: wrap each group in Groups with if-conditions
19: ▷ If WSMode is true but TopRegion is false, if-condition will be added to the caller statement.
20: **return** WSMode
21: **end function**

a thread block. However, CUDA GPUs executes threads in a thread block in an SIMD fashion, meaning all threads in the same thread block executes the same code. To enforce correct semantic of the worker-single mode under the CUDA execution model, each statement in the worker-single mode should be conditionally executed so that only one thread can work on. However, checking conditions for each statement may incur non-trivial overhead. To reduce this overhead, we have developed a transformation technique (algorithm 2), which groups adjacent statements in the worker-single mode together as much as possible, even across the procedure boundary, so that statements in the same group can be guarded by a common condition. In this grouping, a group should be split into two sub-groups at every declaration statement or control-flow statement (e.g., break, continue, etc.), since these should be executed by all threads (line 7 in algorithm 2). The group should also split at each internal barrier annotation implied by the OpenACC semantics (line 8 in algorithm 2).

IV. OpenARC Applications

A. OpenACC Extension to Support Unified Memory

One of the major challenges for the OpenACC programming is to deal with CPU-GPU data movement; most of existing accelerator-based heterogeneous systems have separate address spaces for CPU and GPU, and the cost to communicate data between CPU and GPU is relatively high. For this, OpenACC provides a rich set of data clauses and runtime library routines, but OpenACC still relies on programmers to orchestrate memory management. If programmers do not explicitly guide the data transfers using data clauses, OpenACC offers a default management scheme, which copies all data accessed in a compute region from CPU to GPU before the corresponding GPU kernel is launched and copies back from GPU to CPU after the kernel finishes. The default scheme makes OpenACC programming much easier; Listing 1 shows such an example OpenACC code, where a programmer specifies only compute regions to be offloaded (line 5 and 11 in Listing 1). Then, the underlying OpenACC compiler will generate codes that copy data at each kernel boundary. However, the naive default scheme may suffer from excessive redundant data transfers, as we shall see later in Sect. V-D. Listing 2 shows the case where the programmer manually optimizes data transfers using data directives (line 4 in Listing 2), which dictate that only data *b* should be copied from CPU to GPU at the beginning of the attached data region (line 4) and copied back from GPU to CPU at the end of the attached data region (line 10), reducing the number of memory transfers significantly. However, the manual memory management can be complex and error-prone if a program contains many kernels called across different function boundaries, and/or there are complex user data that requires deep copy of nested data structures.

Unified memory, recently introduced by NVIDIA CUDA 6 and AMD Accelerated Processing Units (APUs), allows that both CPU and GPU access data using a single pointer, and data transfers between CPU and GPU are automatically handled by the underlying system, which dramatically simplifies memory management in GPU programming. OpenACC supports systems with GPUs that have distinct memory from the host CPU, as well as systems with GPUs that share memory with the CPU. In the latter case, an OpenACC implementation may avoid the memory allocation and data movement and simply share the data in the shared memory. This approach will work well if the whole memory is shared by default. However, most of existing GPUs support unified memory by creating a pool of managed memory that is shared between CPU and GPU, which is mainly to minimize the performance tradeoffs; for example, physically fusing two distinct memories (e.g., AMD APUs) needs to be balanced between bandwidth-oriented GPU architectures and latency-oriented CPU architectures, and software-managed unified memory (e.g., NVIDIA CUDA 6) may suffer from large runtime overheads. Therefore, most of existing unified-memory systems require that data should be allocated in a special way to be shared between CPU and GPU, and it is recommended to use the unified memory selectively depending their access patterns. To tightly incorporate these issues into the OpenACC model, we experimentally extended the OpenACC standard using the OpenARC framework. Table I describe the changes in the OpenACC runtime routines. The newly added library routines are used to manage unified memory, and they fall back to CPU memory calls if they are called on a GPU without unified-memory support. All the existing library routines and internal runtime routines used to implement OpenACC data clauses will simply check whether the input data is on the unified memory, skipping any management on the unified memory. Having a separate set of library routines dedicated to manage the unified memory, which can fall back to traditional CPU-memory-

management calls on separate-memory GPUs, allows to create hybrid OpenACC programs that selectively combine separate memory and unified memory, in addition to programs that are fully compatible with both unified memory and separate memory. Listing 3 shows a unified-memory version of the code in Listing 1, where CPU *malloc/free* calls are replaced with the proposed *acc_create_unified/acc_delete_unified* calls (line 1-4 and 14-15), and a data directive with a *present* data clause is added (line 6). (A similar type of code can be created using any standard OpenACC implementation if it supports unified memory.) Even though this version can be created easily by adding a few lines of codes, without analyzing detailed data access patterns to find when to move data between CPU and GPU as in Listing 2, it will work only on unified-memory systems. Listing 4, on the other hand, will work on both unified-memory systems and separate-memory systems; on separate-memory systems, unified-memory-management calls (*acc_create_unified* in line 4 and *acc_delete_unified* in line 14-15) will be reverted back to CPU *malloc/free* calls, resulting in the same behavior as codes in Listing 2. On unified-memory systems, internal runtime routines generated to handle the *copy* data clause will only check the data presence on the unified memory since the argument *b* is allocated on the unified memory (line 4), while the runtime routines for *create* data clause will work as expected. This selective application of the unified memory will endow programmers opportunities to do the precise data orchestration and coordination between CPU and GPU still at high level, resulting in efficient performance as evidenced in Sect. V-D.

Listing 1: Unoptimized Separate-Memory Example

```
1  float (*a)[N2]=(float(*)[N2])malloc(..);
2  float (*b)[N2]=(float(*)[N2])malloc(..);
3  ...
4  for (k = 0; k < ITER; k++) {
5  #pragma acc kernels loop independent
6    for (i = 1; i <= N; i++) {
7      for (j = 1; j <= N; j++) {
8        a[i][j]=(b[i-1][j]+b[i+1][j]+
9        b[i][j-1]+b[i][j+1])/4.0f;
10   } }//kernel-loop1
11 #pragma acc kernels loop independent
12   for (i = 1; i <= N; i++)
13     for (j = 1; j <= N; j++)
14       b[i][j] = a[i][j];//kernel-loop2
15 } //end of k-loop
16 ... //b is accessed by CPU
17 free(a); free(b);
```

Listing 2: Optimized Separate-Memory Example

```
1  float (*a)[N2]=(float(*)[N2])malloc(..);
2  float (*b)[N2]=(float(*)[N2])malloc(..);
3  ...
4  #pragma acc data copy(b), create(a)
5  for (k = 0; k < ITER; k++) {
6  #pragma acc kernels loop independent
7    ...//kernel-loop1
8  #pragma acc kernels loop independent
9    ...//kernel-loop2
10 } //end of k-loop
11 ... //b is accessed by CPU
12 free(a); free(b);
```

TABLE I: Augmented OpenACC Runtime Routines to support Unified-Memory

Runtime Routine	Description
acc_create_unified (pointer, size)	Allocate unified memory if supported; otherwise, allocate CPU memory using *malloc* if input pointer is NULL.
acc_pcreate_unified (pointer, size)	Same as *acc_create_unified* if input data not present on the unified memory; otherwise, do nothing.
acc_copyin_unified (pointer, size)	Allocate unified memory and copy data from the input pointer if supported; otherwise, allocate CPU memory using *malloc* and copy data from the input pointer.
acc_pcopyin_unified (pointer, size)	Same as *acc_copyin_unified* if input data not present on the unified memory; otherwise, do nothing.
acc_delete_unified (pointer, size)	Deallocate memory, which can be either unified memory or CPU memory.
Existing runtime library routines and internal routines used for data clauses	Check whether the input data is on the unified memory; if not on the unified memory, perform the intended operations.

Listing 3: Unified-Memory Example

```
1  float (*a)[N2]=
2    (float(*)[N2])acc_create_unified(..);
3  float (*b)[N2]=
4    (float(*)[N2])acc_create_unified(..);
5  ...
6  #pragma acc data present(a, b)
7  for (k = 0; k < ITER; k++) {
8  #pragma acc kernels loop independent
9    ...//kernel-loop1
10 #pragma acc kernels loop independent
11   ...//kernel-loop2
12 } //end of k-loop
13 ... //b is accessed by CPU
14 acc_delete_unified(a,...);
15 acc_delete_unified(b,...);
```

Listing 4: Hybrid Example that selectively combines both Separate and Unified Memories

```
1  float (*a)[N2]=
2    (float(*)[N2])malloc(..);
3  float (*b)[N2]=
4    (float(*)[N2])acc_create_unified(..);
5  ...
6  #pragma acc data copy(b), create(a)
7  for (k = 0; k < ITER; k++) {
8  #pragma acc kernels loop independent
9    ...//kernel-loop1
10 #pragma acc kernels loop independent
11   ...//kernel-loop2
12 } //end of k-loop
13 ... //b is accessed by CPU
14 acc_delete_unified(a,...);
15 acc_delete_unified(b,...);
```

B. OpenACC Extension to Support Accelerator-Specific Features

OpenACC provides a rich set of directives to guide parallel loop mapping and data sharing. However, it does not provide ones for compiler-specific or architecture-specific information, making it hard to achieve optimal performance. For this, we experimentally extended the OpenACC standard, called *OpenACC-e*, which enables advanced interactions either between compiler passes or between programmers and compilers. The proposed extensions primarily focus on CUDA GPUs, but most of the proposed extensions are also applicable to other types of accelerators. Device-aware OpenACC extension (OpenACC-e) can be grouped with the following categories:

TABLE II: OpenACC Directive Extension for CUDA-Specific Memories

```
#pragma openarc cuda [list of clauses]
where clause is one of the followings:
 global constant, noconstant, texture,
notexture, sharedRO, sharedRW, noshared,
registerRO, registerRW, noregister
```

1) Device-specific memory architecture: OpenACC memory model distinguishes the device memory from the host memory, but it is oblivious of the device sub-memory architectures for portability. In low-level device programming models such as CUDA and OpenCL, programmers can manage device-specific memories/caches, but OpenACC only allows programmers to give hints to the compiler in the form of directives; actual cache/memory managements are up to the compiler. However, this inability for users to manage device-specific memories often incurs a noticeable performance gap between OpenACC and low-level programming models. To address this issue, we extend OpenACC with device-specific directives; Table II shows extensions for CUDA GPUs. (We use a different pragma name (*openarc*) to distinguish these from standard OpenACC directives.) These additional clauses are used to control how the underlying compiler to put specified data in the GPU memory; e.g., *global* clause enforces the compiler to allocate specified data on the CUDA global memory, which may be necessary due to the capacity limit of other special memories, and *sharedRO* directs the compiler to allocate target data on the CUDA *shared* memory. (The compiler may be able to perform additional optimizations since the data are read-only.) We also have clauses to prevent the compiler from using some memories for the target data (e.g., *noconstant*, *notexture*, and *noregister*). These special clauses are mainly used by internal compiler passes to communicate analysis results, but programmers can also use these to fix wrong or inefficient optimization decisions by the compiler.

These directive extensions can still keep enough abstraction over the underlying architectures and also preserve the portability, since ignoring these does not break the original program semantics. However, to fully exploit device-specific memories using directive models like OpenACC, the directive model may also have to provide a more fine-grained control over accessing these memories, described in the following section.

2) Multi-dimensional work-sharing loop mapping: Accelerator devices may support multiple levels of parallelism, and each device may have different types of memories that prefer different access patterns for optimal performance. Moreover, there can be complex interactions among limited hardware resources. Therefore, mapping of work-sharing loop iterations onto device execution units dictates overall resource utilization. For this, OpenACC supports three levels of parallelism (*gang*, *worker*, and *vector* loops), but actual mapping of these work-sharing loops onto the device is up to the compiler. Moreover, the latest OpenACC standard (V2.0) does not allow nested work-sharing loops of the same type (e.g., nested gang loops) except for nested compute regions, which puts significant limits on controlling multi-dimensional work-sharing loop mapping. For example, the CUDA model allows multi-dimensional grid and thread blocks, but with OpenACC, programmer can not explicitly express multi-dimensional grid of gangs or multi-dimensional workers in a gang with specific dimension sizes. (The *tile* clause in OpenACC V2.0 allows multi-dimensional mappings, but the iteration-to-thread mapping is still implicit.) To address this limit, OpenARC extends OpenACC to allow nested work-sharing loops of the same type, if they are tightly nested, and OpenARC applies static mapping for the tightly-nested work-sharing loops. (e.g., the innermost gang loop is statically mapped to the innermost thread block of the CUDA grid.) This static mapping allows programmers to control iteration-to-thread mapping, enabling to manipulate complex thread-access patterns. Combined with additional directives for device-specific memories, this offers an alternative method for programmers to utilize special device memories in a fine-grained manner at OpenACC level (e.g., complex software caching using the CUDA *shared* memory).

This extension can be easily reverted back to the OpenACC standard by adding OpenACC *collapse* clauses to the nested work-sharing loops. If user annotates a loop with *independent* clauses instead of work-sharing clauses (*gang*/*worker* /*vector*), it will let the compiler automatically decide appropriate mappings.

Listing 5: Matrix Multiplication Code in OpenACC-e

```
1  #pragma acc kernels loop gang(N/BSIZE) \
2  copy(C[0:N*N])copyin(A[0:N*N],B[0:N*N])
3  #pragma openarc cuda sharedRW(As, Bs)
4  for(by = 0; by < (N/BSIZE); by++) {
5  //by is mapped to blockIdx.y
6  #pragma acc loop gang(N/BSIZE)
7  for(bx = 0; bx < (N/BSIZE); bx++) {
8  //bx is mapped to blockIdx.x
9      float As[BSIZE][BSIZE];
10     float Bs[BSIZE][BSIZE];
11 #pragma acc loop worker(BSIZE)
12     for(ty = 0; ty<BSIZE; ty++) {
13   //ty is mapped to threadIdx.y
14 #pragma acc loop worker(BSIZE)
15     for(tx = 0; tx<BSIZE; tx++) {
16   //tx is mapped to threadIdx.x
17         int aBegin = wA*BSIZE*by;
18         int aEnd = aBegin+wA-1;
19         int aStep  = BSIZE;
20         int bBegin = BSIZE*bx;
21         int bStep  = BSIZE*wB;
22         float Csub = 0;
23         for (int a=aBegin, b=bBegin; \
```

```
24          a<=aEnd; a+=aStep, b+=bStep) {
25              As[ty][tx] = A[a+wA*ty+tx];
26              Bs[ty][tx] = B[b+wB*ty+tx];
27  #pragma acc barrier
28              for (int k=0; k<BSIZE; ++k)
29                Csub+=As[ty][k]*Bs[k][tx];
30  #pragma acc barrier
31          }; int c = wB*BSIZE*by+BSIZE*bx;
32          C[c + wB * ty + tx] = Csub;
33      } }
34  } }
```

3) Fine-grained synchronization: To handle concurrency and synchronization issues more efficiently, OpenARC extends OpenACC further with a barrier directive (*#pragma acc barrier*) to enable localized and hierarchical implementation that may utilize underlying hardware supports to reduce the overall overhead incurred during these operations. The new barrier directive is allowed only in *worker/vector* loops and has a local scope (a barrier in a worker loop enforces synchronization only among workers in the same gang, and a barrier in a vector loop enforces synchronization only among vector lanes in the same worker). Having this local barriers will enable to port a braided code structure, where the outermost parallel loop consists of a set of inner parallel loops and sequential operations, into a single device kernel, which can save the overhead of additional kernel launches that would incur otherwise [20]. Because the OpenACC execution model assumes implicit barriers at the end of *worker/vector* loops, most of the OpenACC-supporting devices may be able to support this local synchronization. If not, the local synchronization can be achieved via a global synchronization at the end of a kernel, which means that the kernel should be split into two sub-kernels at each barrier; the resulting codes will be compatible with the OpenACC standard. For this, OpenARC supports automatic kernel-splitting transformation. (The kernel-splitting may not always work if the splitting incurs upward-exposed private variables; in this case, OpenARC exits with an error message requesting programmers to manually change the code not to use the local barriers.)

4) OpenACC-e Example: Listing 5 presents an example matrix multiplication code written with OpenACC-e, which has the same computation patterns as those in the example CUDA code of *Matrix Multiplication with Shared Memory* in the CUDA C programming guide [13]. The static mapping of the nested gang/worker loops, where the numbers of gangs/-workers are set to the corresponding loop iteration sizes (line 1, 6, 11, and 14), will ensure that the loop index variables (*bx*, *by*, *tx*, and *ty*) can be used to refer to thread blocks (e.g., use *bx* as if *blockIdx.x* in CUDA)) and threads in a thread block (e.g., use *tx* as if *threadIdx.x* in CUDA)) when ported to CUDA GPUs. The *sharedRW* clause (line 3) is used to allocate gang-private variables (*As* and *Bs* on line 9 and 10) on CUDA shared memory . Combined with the new barrier directives (line 27 and 30), these OpenACC-e features allow users a fine-grained control over thread mapping and software caching with special memories, giving performance comparable to the manual CUDA versions, as shown in Sect. V.

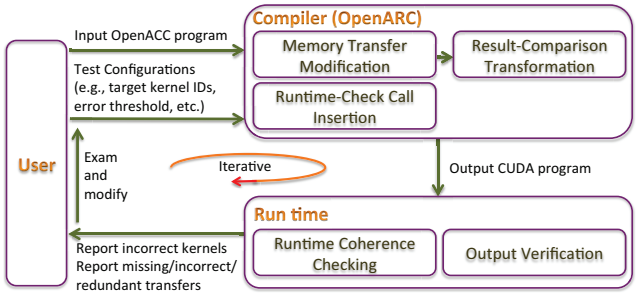


Fig. 1: Interactive GPU Program Debugging and Optimization Framework

C. Interactive Program Debugging and Optimization using OpenARC

To improve debuggability of the directive-based heterogeneous programming models, we must have a systematic approach that exposes more debugging information to programmers, but still at high-level. First, we need more intuitive methods to narrow down the erroneous code regions. Second, we need a traceability mechanism to attribute errors and performance issues back to input directive programs. For these goals, we have developed directive-based, interactive program debugging techniques, which provide better interaction among programmers, compilers, and runtimes [8]. Fig. 1 shows the overall framework. In the proposed framework, the compiler (OpenARC) automatically generates runtime codes necessary for *GPU-kernel verification* and *memory-transfer verification and optimization*. The *GPU-kernel verification* scheme, which consists of compile-time code changes (*Memory Transfer Modification* and *Result-Comparison Transformation* in Fig. 1) and runtime checking (*Output Verification* in Fig. 1), locates trouble-making kernels by comparing execution results at kernel granularity. Figure 2 shows the overall kernel verification process; for each target kernel to verify, 1) temporary buffers are created to keep the kernel execution results, 2) both the kernel and the original compute region are executed on GPU and CPU respectively, and 3) the GPU outputs are compared against the CPU output to verify the correctness of the translated GPU kernel.

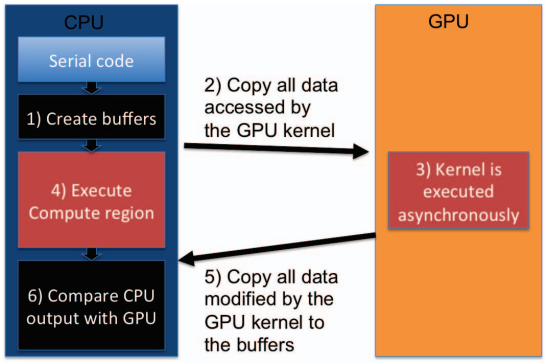


Fig. 2: Overall Kernel Verification Process

The *Memory-transfer verification and optimization* scheme, which consists of compile-time code change (*Runtime-Check*

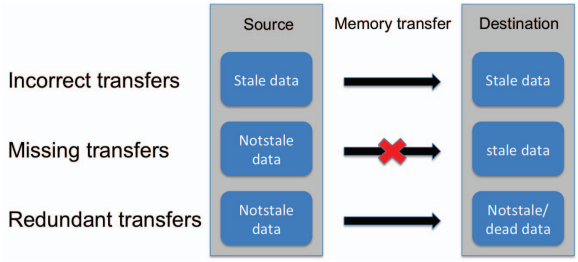


Fig. 3: Runtime Coherence Checking to Detect Incorrect, Missing, or Redundant Memory Transfer

Call Insertion in Fig. 1) and runtime checking (*Runtime Coherence Checking* in Fig. 1), traces runtime status of CPU-GPU coherence to detect incorrect/missing/redundant memory transfers. Figure 3 shows the basic idea to detect incorrect/missing/redundant memory transfers; a memory transfer is 1) *incorrect* if the source is stale, 2) *missing* if the status of the data to access is stale while the other device has up-to-date data, and 3) *redundant* if the destination data is either up-to-date or dead (not used any more). Figure 4 illustrates

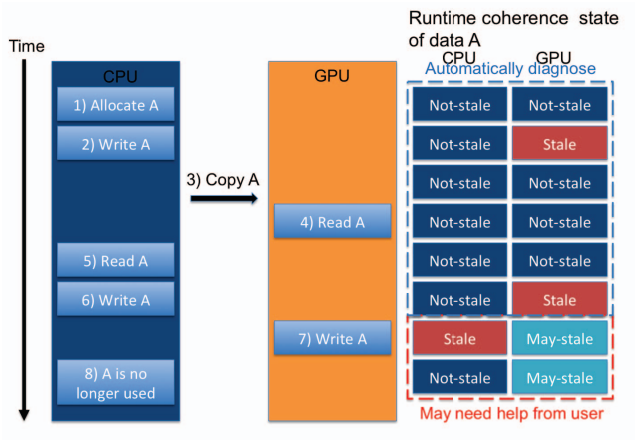
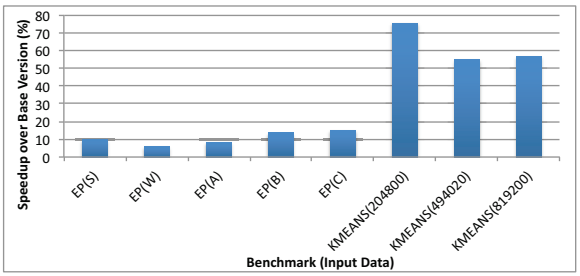


Fig. 4: Runtime Coherence Tracking Example

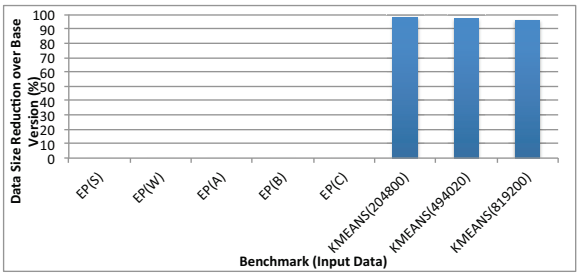
the runtime coherence tracking; when a user data is created, the coherence states of CPU and GPU are initialized to *not-stale* (step 1). If data is modified on a device, the state of the other device should be set to *stale* (step 2, 6, 7). When the data is transferred from one device to the other, the runtime can detect whether the transfer is redundant or incorrect (step 3). For each data access (step 2, 4, 5, 6, 7), the runtime can detect missing transfers if the data to access is stale. However, we may not always need memory transfers to update the stale data if the data is either overwritten (step 7) or no more used (step 8). In these cases, the compiler or runtime may not be able to decide whether it is safe to skip the memory transfers. which requires user involvement. Therefore, the proposed framework is designed as an iterative and interactive system, where users interact with both the compiler and runtime by iteratively fixing/optimizing incorrect kernels/memory transfers, based on the runtime feedback, and applying the updates to the input program via directives. However, a naive implementation of the coherence-based *memory-transfer verification and optimization* scheme may suffer from large runtime checking overhead. Moreover, a naive comparison of the *GPU-kernel verification* scheme may not work due to inconsistent floating-point precision between CPU and GPU. To address these issues, we have developed several optimization techniques, which can be found in the companion paper [8].

V. Evaluation

In this section, we evaluate the OpenARC framework by porting thirteen OpenACC programs from diverse application domains (two NAS Parallel Benchmarks (*CG* and *EP*), three kernel benchmarks (*JACOBI*, *MATMUL* and *SPMUL*), and eight Rodinia Benchmarks [1] (*BACKPROP*, *BFS*, *CFD*, *HOTSPOT*, *KMEANS*, *LUD*, *NW*, *SRAD*)) to CUDA GPUs. For evaluation, two GPU systems are used; a platform with a NVIDIA Tesla M2090 GPU and Intel Xeon X5600 host CPUs is used to compare performance of OpenARC and the PGI OpenACC compiler V13.6 (To compile OpenARC-generated output CUDA programs, NVCC V5.0 and GCC V4.4.6 are used.) To evaluate unified memory versions, the other platform with a NVIDIA Tesla K40c and Intel Xeon E5520 was used, which has NVCC V6.5 and GCC V4.4.7.



(a) Performance Improvement



(b) Memory Transfer Size Reduction

Fig. 5: Effect of the GPU Array Reduction on NVIDIA Tesla M2090. The base version of *EP* uses the array decomposition method, and that of *KMEANS* uses the CPU array reduction.(Both base versions and GPU array reduction versions are translated by OpenARC.)

A. Effect of the GPU Array Reduction

This section demonstrates the effectiveness of OpenARC as a research framework by showing an example study on the effects of array reductions on the accelerator programming; OpenARC supports an array reduction transformation as an experimental feature to OpenACC, which performs a two-level

tree reduction: a local parallel reduction within each thread block, followed by a host-side global reduction across thread blocks. Among the tested benchmarks, both *EP* and *KMEANS* have array reduction patterns. Because the current OpenACC standard does not allow array variables in a *reduction* clause, those GPU array reductions had to be manually modified to alternative patterns to be compatible with the OpenACC standard. One way to replace the GPU array reduction is to decompose it into a set of scalar reductions, which is feasible if the length of the reduction array is short. The decomposition method incurs more control-flow divergence than the GPU array reduction due to conditional updates on scalar reduction variables. The base OpenACC version of *EP* uses this method. The other way is to let the CPU perform the array reduction (CPU array reduction). In this method, each GPU thread works on a local copy of the reduction array, and their local reduction results should be transferred to the CPU so that the CPU performs a global reduction on those local results, which demands more memory transfers than the GPU array reduction. The base OpenACC version of *KMEANS* uses this method. Fig. 5b shows that the decomposition method (*EP*) does not incur additional memory transfer overhead compared to the GPU array reduction. However, *EP* in Fig. 5a indicates that the decomposition method performs slightly worse than the GPU array reduction, mainly due to the additional control-flow divergence. *KMEANS* results suggest that the CPU array reduction may incur non-trivial memory transfer overhead (*KMEANS* in Fig. 5b), performing worse than the GPU array reduction (*KMEANS* in Fig. 5a). The outperforming results of the GPU array reduction in Fig. 5 suggests that adding the array reduction feature to the OpenACC standard may be necessary for both code portability and performance.

B. OpenARC Performance

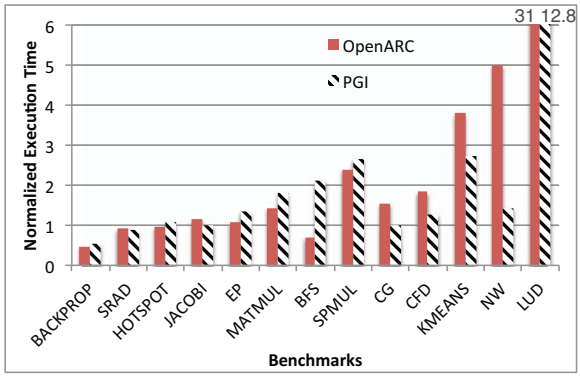


Fig. 6: Performance of OpenACC benchmarks ported by OpenARC and by the PGI compiler on NVIDIA Tesla M2090. The execution times are normalized to those of hand-written CUDA versions. Lower is better.

Fig. 6 presents the performance of the OpenACC benchmarks ported by the OpenARC compiler and the PGI compiler. In the figure, execution times are normalized to those of manual CUDA versions. For the benchmarks, to which corresponding CUDA versions do not exist (*JACOBI*, *SPMUL*, *EP*, and *CG*), locally developed CUDA versions were used. The figure shows that on average, OpenARC-ported OpenACC programs perform similarly to those ported by the PGI compiler, even though OpenARC differs from the PGI compiler in several ways; first, OpenARC allows more fine-grained control over the overall translation and optimization processes, while the PGI compiler implicitly performs most of optimizations, and the user has little control over the translation. Second, very high-level IRs in OpenARC allow to generate output CUDA codes that are much more readable than those generated by the PGI compiler, allowing more debuggability. On the other hand, the PGI compiler provides more advanced optimizations than OpenARC, such as an automatic tiling transformation to exploit CUDA shared memory, which was the main reason for the better performance of the PGI-generated versions in some benchmarks.

The figure also indicates that the OpenACC programs that are automatically ported by OpenARC and the PGI compiler show performance compatible with the manual CUDA versions in many cases. In the figure, normalized values less than one (e.g., *BACKPROP* and *SRAD*) indicate that compiler-generated versions perform better than the existing hand-written CUDA versions. This unexpected worse performance of the manual CUDA programs is mainly due to too much synchronization overhead; the manual CUDA versions of *BACKPROP* and *SRAD* heavily use CUDA shared memory, which requires explicit synchronizations to keep the shared memory consistency. However, the additional synchronization overheads overwhelm the benefits of the shared memory caching. (Hardware caching supported by the tested GPU also diminishes the relative benefit of the manual caching.) However, the excellent performance of the manual CUDA version of *LUD* demonstrates the importance of complex manual optimizations. The manual version of LUD partitions the input data into many subsections and applies different software-caching techniques to each subsection using complex thread-access patterns. The standard OpenACC model does not provide a way to express these complex data access patterns to exploit the CUDA shared memory. However, extensions proposed in Sect. IV-B make it possible to express those complex computations still in the OpenACC codes, as shown in the following section.

C. OpenACC-e Performance

Table III compares the performance of *LUD*, *NW*, and *MATMUL* ported by OpenARC and the PGI compiler, where OpenACC-e refers to the extended OpenACC version translated by OpenARC. The table shows that OpenACC-e could achieve nearly identical performance as the manual CUDA version, demonstrating that the proposed directive extension can effectively exploit architecture-specific features. However, the proposed extension also exposes architectural details of the target device to some extent, while it still hides complex language syntax of low-level programming models. Moreover, the static mapping required by the extended model may restrict the implicit optimizations that the underlying compiler can perform. This trade-off raises a question of right balance between productivity and performance. To answer this question, more in-depth study on the directive-based programming will be needed.

TABLE III: Execution Time of *LUD*, *NW*, and *MATMUL* Normalized to Manual CUDA Version

Benchmark	PGI	OpenARC	OpenACC-e
LUD	12.8	31	1.3
NW	1.4	5	1.1
MATMUL	1.8	1.4	1

D. OpenACC Performance on Unified Memory

Figure 7 presents the performance of OpenACC versions that utilize unified memory (*Unified-Memory*) and OpenACC versions where memory transfers are explicitly optimized by programmers using data directives (*Separate-Memory*). The figure indicates that the unified memory provides both programmability and performance in many cases; except for *BACKPROP*, *EP*, and *MATMUL* programs, simply using the unified memory (*Unified-Memory*) could remove more than 90% of their execution times by removing redundant memory transfers. This is because most of tested programs have good localities; most of the tested programs have patterns where data are copied from CPU to GPU at the entrance of the outermost loop and remains in the GPU mostly during the main loop computation. NAS Parallel Benchmark *CG* is a good example demonstrating the power of unified-memory programming. Even though *CG* contains more than thirty kernels called across different function boundaries, difficult to analyze optimal memory transfer patterns, the actual data access patterns have very good localities; most of user data are private to GPU, and only a few data need to be copied in at the entrance of the main computation region. In this case, simply replacing CPU *malloc/free* calls to those for unified-memory management and letting the underlying system manage the data transfers can achieve good performance comparable to manually optimized version (*Separate-Memory*).

JACOBI and *SRAD* are the most interesting cases in that *Unified-Memory* versions perform better than manually optimized OpenACC versions (*Separate-Memory*). The tested *JACOBI* benchmark verifies its computation by comparing diagonal sum of the output array; in the standard OpenACC version for separate-memory systems (*Separate-Memory*), the whole output data should be copied from GPU to CPU after the main computation finishes, since the diagonal sum has sparse accesses throughout the whole array. However, in the OpenACC version utilizing the unified memory, the underlying memory management system copies data from GPU only if CPU requests (*full lazy copy*), incurring much smaller amount of data transfers than the *Separate-Memory* version. In *SRAD*, a small subset of a logically two-dimensional array (*J*) are copied from GPU to CPU at each iteration of the outermost loop, but the location of the subset is decided at runtime, and the data subset are discontinuous in the memory layout. Therefore, the standard OpenACC version for separate-memory systems copies the whole array (*J*) at each iteration. However, the unified-memory version can copy only the accessed subset thanks to the *full lazy copy* feature. Moreover, the main loop is executed multiple times, resulting in much better performance than the separate-memory version. *JACOBI* and *SRAD* cases indicate that if only small random portion of data are accessed, using the unified memory may perform much better than the manual memory transfers using data clauses, demonstrating that the hybrid scheme, proposed in Sect. IV-A, that combines both unified-memory and separate-memory together can be very effective on such types of applications.

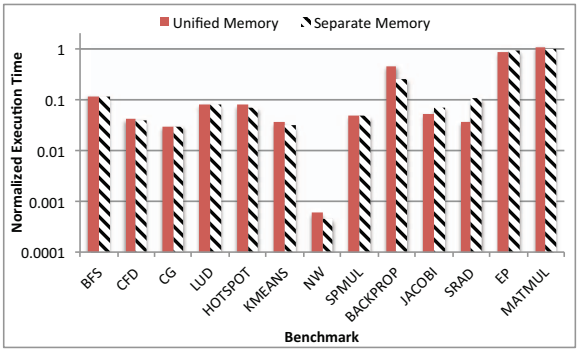


Fig. 7: Performance of OpenACC versions utilizing unified memory (*Unified-Memory*) and OpenACC versions where programmers explicitly control memory transfers using data directives (*Separate-Memory*) on NVIDIA Tesla K40c. Execution times are normalized to those of the naive, default memory management scheme (no memory-transfer optimization).

VI. Conclusions

In this paper, we have presented an extensible OpenACC compiler framework, called OpenARC. OpenARC is equipped with various analysis and transformation tools, serving as a powerful research framework to study various issues in directive-based, heterogeneous computing, such as debuggability, performance portability, tunability, scalability, etc. This paper provides an overview of important design strategies and several key techniques needed to implement a reference OpenACC compiler. In addition, we demonstrated the efficacy of OpenARC as a general research framework for directive-based, high-level programming investigations by providing examples of such prototypes; these examples include OpenACC extension to support hybrid programming of the unified memory and separate memory and device-aware OpenACC extensions to express architecture-specific features at high-level. Example porting of three benchmarks (*LUD*, *NW*, and *MATMUL*), using the extended OpenACC model provides an insight on the right scope of the interaction between programming models and compilers/runtimes. Porting thirteen standard OpenACC benchmarks from diverse application domains shows that OpenARC performs similarly to a commercial compiler, while OpenARC provides more fine-grained control over the overall translation and optimizations, suitable for various performance optimization studies.

Acknowledgment

The paper was authored by Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract DE-AC05-00OR22725 to the U.S. Government. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. This research is sponsored by the Office of Advanced Scientific Computing Research in the U.S. Department of Energy.

REFERENCES

[1] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. ha Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[2] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *IEEE Computer*, vol. 42, no. 12, pp. 36–42, 2009.

[3] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. van der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick, "The international exascale software project roadmap," *International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, 2011. [Online]. Available: http://hpc.sagepub.com/content/25/1/3.abstract

[4] T. D. Han and T. S. Abdelrahman, "hiCUDA: High-level GPGPU programming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 78–90, 2011.

[5] HMPP, "OpenHMPP directive-based programming model for hybrid computing," [Online]. Available: http://www.caps-entreprise.com/openhmpp-directives/, 2009, (Accessed August 30, 2014).

[6] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," DARPA Information Processing Techniques Office, Tech. Rep., 2008.

[7] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP programming and tuning for GPUs," in *SC'10: Proceedings of the 2010 ACM/IEEE conference on Supercomputing*. IEEE press, 2010.

[8] S. Lee, D. Li, and J. S. Vetter, "Interactive program debugging and optimization for directive-based GPU computing," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, may 2014.

[9] S. Lee and J. Vetter, "OpenARC: Open Accelerator Research Compiler for Directive-Based, Efficient Heterogeneous Computing," in *HPDC '14: Proceedings of the ACM Symposium on High-Performance Parallel and Distributed Computing, Short Paper*, june 2014.

[10] S. Lee and J. S. Vetter, "Early evaluation of directive-based GPU programming models for productive Exascale computing," in *the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. IEEE press, 2012.

[11] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin, "A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. ACM, 2010, pp. 51–61.

[12] C. Liao, Y. Yan, B. Supinski, D. Quinlan, and B. Chapman, "Early experiences with the OpenMP accelerator model," in *OpenMP in the Era of Low Power Devices and Accelerators*, ser. Lecture Notes in Computer Science, vol. 8122. Springer Berlin Heidelberg, 2013, pp. 84–98.

[13] NVIDIA, "CUDA C Programming Guide," [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2013, (Accessed August 30, 2014).

[14] OpenACC, "OpenACC: Directives for Accelerators," [Online]. Available: http://www.openacc-standard.org, 2011, (Accessed August 30, 2014).

[15] OpenMP, "OpenMP: Version 4.0," [Online]. Available: http://openmp.org/wp/openmp-specifications/, 2013, (Accessed August 30, 2014).

[16] PGI_Accelerator, "The Portland Group, PGI Fortran and C Accelarator Programming Model," [Online]. Available: http://www.pgroup.com/resources/accel.htm, 2009, (Accessed August 30, 2014).

[17] D. Quinlan and C. Liao, "The rose source-to-source compiler infrastructure," in *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT 2011*, October 2011.

[18] R. Reyes, I. López-Rodríguez, J. Fumero, and F. Sande, "accULL: An OpenACC implementation with CUDA and OpenCL support," in *Euro-Par 2012 Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 7484. Springer Berlin Heidelberg, 2012, pp. 871–882.

[19] A. Sabne, P. Sakdhnagool, S. Lee, and J. S. Vetter, "Evaluating performance portability of OpenACC," in *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2014.

[20] L. G. Szafaryn, T. Gamblin, B. R. de Supinski, and K. Skadron, "Trellis: Portability across architectures with a high-level framework," *Journal of Parallel and Distributed Computing*, vol. 73, no. 10, pp. 1400 – 1413, 2013.

[21] X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman, "Compiling a high-level directive-based programming model for GPGPUs," *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2013.

[22] J. S. Vetter, Ed., *Contemporary High Performance Computing: From Petascale Toward Exascale*, 1st ed., ser. CRC Computational Science Series. Boca Raton: Taylor and Francis, 2013, vol. 1.