Parallel Computing 78 (2018) 15–27

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

The OpenACC data model: Preliminary study on its major challenges and implementations☆

Michael Wolfe[a,*], Seyong Lee[b], Jungwon Kim[b], Xiaonan Tian[a], Rengan Xu[c], Barbara Chapman[d], Sunita Chandrasekaran[e]

[a] NVIDIA/PGI, USA
[b] Oak Ridge National Laboratory, USA
[c] Dell, USA
[d] Stony Brook University, USA
[e] University of Delaware, USA

ARTICLE INFO

Article history:
Received 18 December 2017
Revised 25 June 2018
Accepted 17 July 2018
Available online 19 July 2018

Keywords:
OpenACC
Data model
Compiler implementations
Accelerators

ABSTRACT

This paper describes how the OpenACC data model is implemented in current OpenACC compilers, ranging from research compilers (OpenUH and OpenARC) to a commercial compiler (the PGI OpenACC compiler). First, we summarize various memory architectures in today's accelerator systems. We then describe details and issues in implementing the OpenACC data model in three different OpenACC compilers. This includes managing page tables, asynchronous data transfers, asynchronous memory allocate and free, host data construct, aliasing on a data directive, reusing device memory, partially present data, and adjacent data. We also discusses ongoing work to manage large, complex dynamic data structures. We measured the present table lookups, device memory allocation, pinned memory allocation, and managed memory in the three OpenACC compilers using eight OpenACC applications (seven from the SPEC ACCEL benchmark suite and a shock-hydrodynamics mini-application called LULESH).

1. Introduction

Accelerator-based heterogeneous computing has been used in high-performance computing for many years in applications ranging from early accelerators such as attached array processors (Floating Point Systems [1], IBM [2], etc.) to recent general-purpose GPUs from NVIDIA and AMD, many-core coprocessors from Intel [3], digital signal processors from Texas Instruments [4], and field-programmable gate arrays. Two important issues in the accelerator-based heterogeneous computing are programmability and portability across diverse architectures. The OpenACC Application Programming Interface [5] is

* Corresponding author.
E-mail addresses: mwolfe@nvidia.com (M. Wolfe), lees2@ornl.gov (S. Lee), kimj@ornl.gov (J. Kim), datian@nvidia.com (X. Tian), Rengan_Xu@dell.com (R. Xu), barbara.chapman@stonybrook.edu (B. Chapman), schandra@udel.edu (S. Chandrasekaran).

https://doi.org/10.1016/j.parco.2018.07.003
0167-8191/

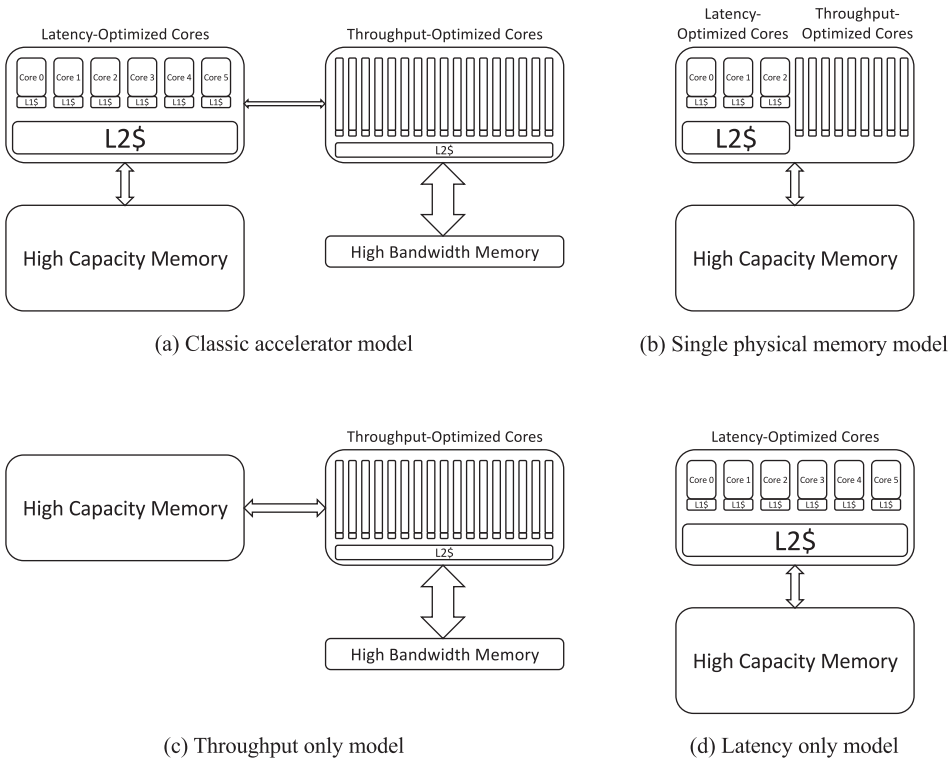

Fig. 1. Target memory architectures.

one of such approaches to answer these issues. OpenACC uses directives, quite similar to the OpenMP API and various other directive sets [6,7], for the user to tell the compiler what data to move and when to move it between the host and device memories, and what computation to perform on the accelerator and what to leave on the host. With currently available accelerators, managing the data movement is the first and perhaps the biggest bottleneck in achieving good performance. Initial attempts to automatically manage the data movement [7–9] worked well in certain simple, stunt examples, but work uniformly poorly in general, leading the developers of OpenACC to allow the programmer to manage this manually.

OpenACC was initially designed to work with accelerator devices with their own memory, as well as devices that share memory with the host. As we look forward to the near future, we anticipate a range of architectures being developed with more interesting memory architectures. The OpenACC data model describes what data gets moved or copied to the device memory or back to the host memory and when that movement or copy is done. Usually the data movement is controlled by OpenACC data directives or clauses. There are several commercial and research implementations of OpenACC [10–14]. This paper describes how three current OpenACC compilers (PGI [10], OpenARC [13], and OpenUH [14]) implement the OpenACC data model. This paper also describes the ongoing work on adding *deep copy* behavior to OpenACC, which is needed to manage large, complex dynamic data structures, and evaluates the different implementations of the OpenACC data model by the tested compilers. This paper is an extended version of a previous work published in the proceedings of the Seventh International Workshop on Accelerators and Hybrid Exascale Systems workshop [15]. Compared to the previous work, this version includes (1) several new sections about host data construct, partially overlapping data, and adjacent data; (2) new evaluation results for pinned memory and managed memory; (3) analysis of a shock-hydrodynamics mini-application called the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). Other sections are heavily updated with new content or have been deleted.

2. Target memory architectures

OpenACC was designed to support multiple different system memory architectures. Fig. 1(a) shows the most common memory architecture that OpenACC supports, called the *classic accelerator* memory architecture. This architecture consists of a host and one or more accelerator devices, where the host is a latency-optimized multicore with a large coherent cache memory attached to a very high capacity memory, and the device consists of many throughput-optimized cores with much smaller cache memory attached to a much smaller but much higher bandwidth memory (HBM). Today, the typical connection between the host and accelerator is through the PCI-express IO bus. To the host multicore and operating system, the accelerator is an IO device controlled by OS drivers. In the most limited case, the host cannot access the HBM directly, and

procedures running on the accelerator cannot access the host memory directly. To use the accelerator requires the program to (1) allocate space in the HBM, (2) copy input data from the host memory to the HBM, (3) launch one or more procedures or kernels on the accelerator to do the calculations, (4) copy results from the HBM back to the host memory, and (5) deallocate the space in the HBM.

The datapath between the host and the accelerator has low bandwidth compared to the host or device memories, so optimizing the program to minimize the frequency of data transfers and amount of data transferred is key to good performance. The OpenACC data directives and clauses allow a programmer to perform this optimization.

NVIDIA Kepler GPUs support a shared address space between the CPU and GPU, called *CUDA Unified Memory* or *managed memory*. When a program allocates managed memory using special allocation routines, the runtime will allocate space in the host memory as well as the HBM. When a compute kernel is launched, the CUDA driver will move any managed memory data from the host memory to the HBM and mark the CPU pages as nonresident. This is necessary because the Kepler GPU does not have page fault detection hardware. A subsequent CPU reference to managed memory space will trigger a page fault on the CPU, and the driver will bring that data back to the CPU side. There are limitations on how this can be used, but some OpenACC implementations are able to use this. Newer NVIDIA Pascal and Volta GPUs support CUDA Unified Memory with fewer limitations. Since these GPUs support page translation, data is paged to the device memory when referenced by a GPU kernel, instead of at the kernel launch.

The latest NVIDIA Volta GPUs and future NVIDIA GPUs will have the same physical layout shown in Fig. 1(a) and a much higher bandwidth connection between the accelerator and the multicore host. Not only will the connection run at almost the same bandwidth as the host memory interface, it will allow coherent accesses from the accelerator to host memory and from the multicore to the HBM. Specifically, it will allow the accelerator to access data in the host memory without being specially allocated and without pinning the data, although there will still be a performance penalty relative to data in the HBM because of the bandwidth differences. Such a feature opens new opportunities and challenges for optimizing OpenACC programs. We call this design *unified virtual memory*. (The latest IBM Power9 system with Volta GPUs preliminary supports the unified virtual memory.) Such a design may allow the runtime or operating system to dynamically choose whether to move data between the physical memories based on recent memory reference patterns or to replicate the data in both memories if it is read-only.

An alternative to the separate memories is exemplified by the AMD Accelerator Processing Unit. In this design, both latency-optimized cores and throughput-optimized cores are included on a single processor chip, both using the same cache hierarchy and same memory interface to the host memory, as shown in Fig. 1(b). This design has several important advantages. Even if the operating system and software do not allow the CPU cores and accelerator cores to share data, the data copies between the host space and accelerator space can be done at full memory bandwidth. If the software does allow the different cores to share data with the same addresses, then data does not need to be be moved at all. This is true shared memory, and this simplifies programming dramatically. However, this comes with its own performance penalty. Since there is no HBM, all data accesses, even from the throughput-optimized cores, will run at the much lower host memory bandwidth. We call this design *single physical memory*.

Another alternative is demonstrated by the latest Intel Knights Landing Xeon Phi processor. Here, the processor only has throughput-optimized cores, which connect directly to both a high-bandwidth memory and a high-capacity memory, as shown in Fig. 1(c). If all the data fits into the HBM, there is no problem, but this is unlikely. In general, data must be moved or copied into the HBM when it is being intensely used, and moved or copied back to the host memory afterward. The OpenACC data directives and clauses can be used as hints to the implementation about when data should be moved to the HBM and back. We call this design *throughput-only*.

To complete the catalog, OpenACC can also be used in a typical multicore configuration, as shown in Fig. 1(d), without an HBM and accelerator. In this case, an OpenACC implementation can ignore most of the data directives and clauses, and generate parallel code more or less like the a corresponding OpenMP implementation. We call this design simply a *latency-only*.

In each case, the programmer or the OpenACC implementation must decide whether and when to copy or move data from the host memory to the HBM and back. We distinguish between copying data and moving data. When copying data from one memory to the other (e.g., from the host memory to the HBM) the original copy still exists in the host memory and is accessible from the host. When moving data from one memory to the other, the data only exists in the target memory.

3. OpenACC data model

The OpenACC execution model is a host-directed execution with one or more attached *accelerators*. OpenACC compute directives are used to offload compute-intensive regions to the accelerator device, and OpenACC data directives and clauses are used to manage the device memory. This section describes the OpenACC data directives and clauses, and the expected behavior on the various targets.

OpenACC has a structured *data construct* that allows a program to tell the implementation when certain data needs to be available on the device. The most important *data clause* is the *copy* clause, which says that if a copy of the data is allocated in device memory, that data must be initialized with the existing values from host memory, and when the device is done processing the data, the final values must be copied back to host memory. Execution of a *data construct* creates a *data region*,

Table 1
Implementations comparison.

Implementation	PGI	OpenARC	OpenUH
Present table	Red–black tree	Red–black tree	Hash table
Asynchronous data transfer	Upload pinned buffer, download pinned buffer	Prepinning, lazy pinning	Lazy pinning
Asynchoronous allocate and free	Sync allocate, *async* free	Sync allocate, *async* free	Sync allocate, *async* free
Host data construct	Support	Support	Support
Aliasing on a data directive	Support	Limited support with full copy	Not support
Reusing device memory	Free device memory pool	Free device memory pool	Not support
Partially overlapping data and adjacent data	Bad data references	Runtime error	Runtime error

which is the dynamic range of the construct. The data specified in the *data construct* clauses will be available on the device over the whole data region, including any procedures called within that region.

OpenACC also includes dynamic or unstructured data lifetimes, with the *enter data* and *exit data* directives. The *enter data* directive acts much like the entry to a structured *data* construct, and the *exit data* directive acts very like the exit from a structured *data construct*.

For some targets, the data directives can be completely ignored. For *latency-only* and *single physical memory* targets, there is only one copy of the data, and the same address is used by host and accelerator cores (if there are accelerator cores). For a *classic accelerator*, data must be allocated and copied to and from device memory because the accelerator cores cannot access host memory. For *managed memory*, the data needs to be allocated in the proper way. OpenACC does not control the memory allocation; the data could be static, global, or local to a procedure (on the stack), or it could be dynamically allocated. We will see some implementations that are experimenting with *managed memory*. For *unified virtual memory* or *throughput-only* targets, the OpenACC runtime will have to decide whether to move the data to the HBM or leave it in the larger main memory.

When data copies must be made, both structured and unstructured data directives are implemented using reference counting. Each block of memory in device memory has two reference counts—one for static *data constructs* and one for dynamic data directives. When a *data construct* or an *enter data* directive specifies some block of data that must be allocated in device memory and is not yet present, that data is created, and the reference counts are both initialized to zero. For entry to a *data construct*, the static reference count for the data block is incremented, whereas for an *enter data* directive, the dynamic reference count is incremented. At exit from a *data construct*, the static reference count is decremented, and at an *exit data* construct, the dynamic reference count is decremented. In either case, if both reference counts reach zero, the data is then deallocated.

4. Implementations

This section describes how the OpenACC data model is implemented in three compilers. PGI provides commercial compilers for C, C++, and Fortran, which include the OpenACC 2.0 directives, OpenMP 3.1 directives, and many other features [10]. The OpenACC directives grew out of the PGI Accelerator directives that were first introduced in 2008. See www.pgroup.com for more information about the PGI compilers.

OpenARC [13] is an open source, extensible research compiler framework based on Cetus [16], which performs source-to-source translation from OpenACC C to CUDA/OpenCL, targeting various architectures from NVIDIA/AMD GPUs to Intel Xeon Phi Coprocessors and Altera field-programmable gate arrays. See ft.ornl.gov/research/openarc for more information about OpenARC.

OpenUH [14,17] is an open source, optimizing compiler suite for C, C++, and Fortran based on Open64, supporting a variety of target architectures. OpenUH is developed and maintained by the High Performance Computing Tools research group at the University of Houston. See github.com/uhhpctools/openuh-openacc for more information about OpenUH.

Table 1 provides a brief comparison of the implementations for the three different compilers.

4.1. The present table

The *present table* is the key data structure in an OpenACC runtime. The present table keeps track of what data is available in device memory. It must be indexed by the host address of the data. Variable names cannot be used since they do not persist when pointers are passed as arguments to a routine, and an implementation cannot add information to a pointer (*fat pointer*) because a user does not necessarily compile all the intermediate routines with the same compiler. The only value that persists across all places where the data is used is the address of the data.

If the program always used data starting at the same point, the *present table* could be implemented with a hash table, indexed by the starting host address. However, C and C++ programs often do pointer arithmetic and Fortran programs often pass subarrays as arguments to other routines, so the present table must be able to accept any address in the range of the data object and return the appropriate device address. Another way to implement the present table is to use a linear list

of data, which is fast enough when the number of data objects is small. However, as usage grows and larger programs are ported, and as deeper data structures are moved to the device, the number of data objects can grow in some cases to the thousands. A typical implementation today uses a red–black tree indexed by the host address range to store the *present table*. A red–black is a reasonably balanced binary tree, so lookup, insert, and delete operations take $O(\log n)$ time.

To handle OpenACC data model, the following operations are typically handled for the present table. At entry to a structured *data construct* or at an *enter data* directive with a *copy, copyin, copyout*, or *create* clause, the program computes the host start and end addresses and does a *present table* lookup. If the data is already present, the appropriate reference count is incremented and the appropriate device address is returned to pass to the kernel function on the device. If the data is not present, it is allocated and (for the *copy* and *copyin* clauses) initialized with the host data. A new *present table* entry is allocated and initialized, and inserted into the table. If the data is partially present, a fatal runtime error is issued; see Section 4.7 for discussion about this issue.

At exit from a structured *data construct* or at an *exit data* directive, the program computes the host start and end addresses again and does another *present table* lookup. The data should be present, or a fatal runtime error is issued. The appropriate reference count is decremented. If both reference counts are zero after the decrement, then (for *copy* and *copyout* clauses) the device data is copied to the corresponding host location, and the device data is deallocated. The *present table* entry is then removed from the table and recycled.

If the implementation supports more than one device, then it must have a *present table* for each device. If the host can be treated as a device, then all data is by definition present for the host device, and no *present table* is needed. If a device shares memory with the host, and there is no performance penalty for doing so, then again all data is by definition present, and a *present table* is not needed. On the other hand, if a device shares memory with the host but with a performance penalty, either the program or the runtime must decide whether to use the data from host memory at lower performance or to incur the overhead of moving the data to the high-bandwidth memory. In this case, a *present table* may be needed to distinguish between data in the host memory and data in the HBM.

4.1.1. PGI

The PGI OpenACC runtime uses a red–black tree indexed by the host address range for the *present table*. The red–black tree–based implementation can accept any address in the range of the data object and return the appropriate device address. The table entry stores the start and end addresses of the corresponding host data, the start address of the device data, and the two reference counts. The PGI implementation stores additional information to help with data inspection. PGI supports multiple devices with a separate table for each device. In a multithreading environment, the runtime uses locks when modifying or searching the table.

The latest PGI compiler has a feature to take advantage of CUDA *managed memory*. For NVIDIA GPUs, the PGI implementation supports a command line flag that will replace C *malloc* calls, C *calloc* calls, C++ *new* invocations, and Fortran *allocate* statements by versions that allocate CUDA-managed memory. The PGI runtime then uses the CUDA API to test whether the host address in a data clause lies in the *managed memory* region. If so, the runtime lets the CUDA driver manage the data.

4.1.2. OpenARC

The OpenARC runtime implements the *present table* using a C++ Standard Template Library map (one map per device), which maps a starting host address to a tuple of a device pointer and the data size. Because C++ STL maps are typically implemented as red–black trees where the elements are always sorted by their keys following a specific strict weak ordering criterion, present table lookups will take $O(\log n)$ time for any address in the range of the data object.

To support CUDA *managed memory*, the OpenARC runtime offers extra OpenACC runtime library routines [13]. These routines allocate and free managed memory and can revert to using system memory if the target device does not support managed memory. The present table also tracks data allocated in managed memory by using the host address for both the table key and device pointer. The OpenARC runtime can check whether host data is allocated in the managed memory region by comparing the host address and the device address. If they are equal, the OpenARC runtime lets the underlying driver manage the data. Having a separate set of managed memory APIs that are backward-compatible with traditional CPU management calls allows hybrid OpenACC programs that selectively use managed memory and execute correctly on systems that do not support managed memory.

4.1.3. OpenUH

The OpenUH compiler uses hash tables for the present table implementation. It maintains static and dynamic hash tables, which are maintained by the compiler and the runtime, respectively. The static hash table is indexed by the symbol table in the compiler. This table is used for structured data constructs at compile time. Within a structured data region, there may be many compute regions typically sharing data. The static hash table helps the compiler to save the device address of the data once for all compute regions within structured data constructs. The dynamic hash table is maintained by the runtime. This table is used for data regions at runtime and can be used for both structured data region and unstructured data directives. Each entry in the dynamic hash table includes the host address, device address, and data size.

The OpenUH runtime maintains a region stack to track the region chain and the new data created within each region. The region stack can guarantee that the data list created at the entry of a region (be it data or compute region) will be freed at the exit of the same region. For example, whenever a new data region is encountered, a region pointer is pushed

into the region stack. If the regions are not nested, then they are pushed into the stack in sequence. All the newly created data in the current region are appended to a linked list and then inserted into the dynamic hash table. The device memory is allocated for these data and copied to the device as necessary. At the exit of a region, the runtime will pop the region pointer from the region stack, copy the data from the device address to the host address (for `copy` and `copyout` clauses) and free the data list created in that region.

For partially present data, OpenUH allocates the device memory for the whole array rather than the partial array size; so for the same data, the host copy and the device copy always have the same size. It is impossible to check the partial data in the hash map simply by hashing because the passed data address is not the start of the host address. So the OpenUH implementation may traverse the entire hash map to check whether the partial data range is within the whole data range.

4.2. Asynchronous data transfers

OpenACC allows a program to specify that a compute kernel or data transfer should execute asynchronously. An *async* operation can be executed on the device while the host program continues until it reaches a *wait* operation that waits for its completion. OpenACC allows the implementation to support several *async* queues on which operations are enqueued, where any operation will not start until all earlier operations on that same queue are complete. On NVIDIA GPUs, a true asynchronous data transfer requires that the host memory is *pinned*, that is, allocated or moved to contiguous physical memory that the operating system will not page. This allows the direct memory access unit to move data with a simple physical start address and length, without interfacing to the operating system page tables. However, OpenACC does not usually have control over memory allocation. Some data may be statically allocated, or a local variable on the stack. Dynamically allocated memory may be allocated by a routine compiled without OpenACC. There is an interface to *pin* a range of memory, but even this is hard to use productively.

There are multiple ways to support asynchronous data transfers using the pinned memory. In the first approach, the host data is pinned when the data is allocated on the device and unpinned when the device data is deallocated. This approach works, but it may suffer from high overhead for unpinning data because unpinning data requires the host to synchronize with the device, to ensure there are no outstanding asynchronous transfers to that pinned memory. In the second approach, the host data is pinned when the data is allocated on the device and that memory is never unpinned. However, this approach often does not work. For dynamically allocated data, memory is automatically unpinned when it is freed, but the OpenACC runtime cannot trap the deallocation event so does not realize that memory range is no longer pinned when it gets reused for a subsequently allocated object. For stack data, the same behavior may occur when the routine returns. Another problem arises when the program has a loop around a data construct or two nonoverlapping data constructs, where the second data construct entry copies a larger or intersecting block of memory to the device. In that event, the runtime wants to ensure that the additional pinned memory is pinned and located adjacent to the original pinned memory.

4.2.1. PGI

The default PGI implementation for NVIDIA GPUs is to use two pinned buffers for upload and download, which allows for asynchronous transfers without pinning user memory. For upload, the runtime will synchronously copy the user memory to the first pinned buffer and issue an asynchronous data transfer. If the data size is larger than the pinned buffer, it will continue using the second buffer, and loop back to the first buffer in a classical double-buffering scheme. For download, the runtime will issue asynchronous data transfers to a download buffer, then save a descriptor with information about where in host memory that data should be copied. When another download needs that buffer or when the host is waiting on the device, the runtime will copy the data from the buffer to the user memory.

4.2.2. OpenARC

For NVIDIA GPUs, the OpenARC runtime offers two modes: *prepinning* and *lazy pinning*. In the prepinning mode, the host data is always pinned when the data is allocated on the device because the runtime may not know whether this data will be transferred asynchronously or not at the GPU memory allocation time. In the lazy pinning mode, on the other hand, the host data is pinned just before the first asynchronous transfer. This mode pins the host data only if they are transferred asynchronously, saving unnecessary memory pinning, but the runtime should track each asynchronous transfer to determine the right time for pinning.

4.2.3. OpenUH

In the OpenUH implementation, the structure of the data not only includes the host address, device address, and data size, but it also includes a *pinned* flag, which is initialized to zero. When the data is transferred asynchronously, if the *pinned* flag is not set, the runtime will pin the whole host memory region for that data, regardless of the transfer size, and set the *pinned* flag to one.

4.3. Asynchronous allocate and free

At entry to a *data* construct or at an *enter data* directive, device memory may need to be allocated. At exit from a *data* construct or at an *exit data* directive, device memory may need to be freed. Does an implementation need to support

```
void sub( float* x, float* y, int n ){
    #pragma acc data copyin(x[0:n]) copyout(y[0:n])
    {  ...  }
}
float* a; ...; sub( a, a, n );
```

Fig. 2. Data aliasing example.

asynchronous data allocate and free? Even if device memory allocation could be done asynchronously, the device address can be captured by the program. This means the address must be assigned synchronously. At exit from a *data* construct or at an *exit data* directive, any *copyout* operations must complete before the memory is freed for reuse. If the free operation is synchronous, the data movement for that memory must be synchronous too.

4.3.1. PGI

The PGI implementation allows for asynchronous free operations. As described earlier, the runtime uses asynchronous data transfers to internal pinned buffers. The runtime saves a descriptor, so that at some later point, such as at a synchronization or when the buffer is needed, the runtime can copy the data from the buffer to the user memory. The descriptor also saves whether the device memory should be freed. This means that the device memory is not freed until all transfers are complete.

In theory, at an asynchronous allocate, the runtime could search through the *async* queue for an asynchronous free, and reuse that address for the allocate. This would assign an address, and as long as the data is only accessed on this queue or after a synchronization, it will be safe. The PGI runtime does not do this, however. Instead, it implements allocate operations synchronously. If the allocate fails because the device memory is full and there are outstanding asynchronous downloads pending, the runtime will drain the download queues searching for pending memory free operations.

4.3.2. OpenARC

The OpenARC implementation allows asynchronous free operations, while allocations are synchronous. In the OpenARC implementation, the host data to be asynchronously transferred are always pinned, either eagerly or lazily, and thus additional copy between the internal buffer and the host data is unnecessary. Asynchronous free operations are implemented by putting the host address and the associated *async* queue in a *postponed-free* table. Synchronization operations check the *postponed-free* table and perform pending free operations associated with the synchronizations.

4.3.3. OpenUH

In OpenUH, the runtime creates the same number of host helper threads as the number of devices. Each thread is associated with one device. Each thread creates an empty first in first out task queue that will be populated by the host main thread. The main thread enqueues operations onto the appropriate task queue. The corresponding helper thread can then asynchronously perform memory allocate, deallocate, data transfer, and kernel launch operations. However, memory allocation will behave synchronously and the main thread will wait for the helper thread to provide the device address.

4.4. The host data construct

OpenACC includes a *host data* construct, which replaces references to an object by references using the device address of that object. This is typically used to interoperate with a device's native API; the following example could be used to pass a device address to a hand-tuned CUDA kernel or to a GPU-aware MPI library:

```
#pragma acc host_data use_device(a[0 : n])
{     device_optimized(a, n);     }
```

Implementing *host data* simply requires indexing into the *present table* and returning the associated device address. There is occasionally a need for a program to obtain the host address associated with a device address, but this may result in nontrivial overhead because the *present table* is indexed by host address, not device address. The three tested compilers implement an API to do this simply by walking the entire *present table* until the associated device address is found, assuming that the use case is rare enough.

4.5. Aliasing on a data directive

When two or more objects specified on a data directive are aliased with each other, OpenACC 2.0 is silent on the intended behavior. In Fig. 2, the programmer would expect the array a to be allocated on the device, the data to be copied in at the top and copied out at the bottom. However, not all implementations do this. The latest OpenACC 2.5 specification defines the behavior that all data movement associated with aliased data on the same directive must be honored. A safe implementation could copy all data in at entry and copy all data out at exit from a data lifetime, but optimizing this data movement can be important for performance.

4.5.1. PGI

In the PGI implementation, the data clauses are processed left-to-right at the top of a data construct. In Fig. 2, the data will be allocated and copied in to the device for the first clause. When the second clause is reached, the runtime will find that the array y is already present, so its reference count will be incremented. At the end of the data construct, the clauses are processed right-to-left. The second clause will decrement the reference count and find it is not yet zero, so will not copy the data out. The first clause will then decrement the reference count to zero, and because it is a *copyin* clause, will simply free the array.

4.5.2. OpenARC

The OpenARC implementation uses reference counts to control memory allocation and free operations, but it always performs the data movements of data clauses regardless of their reference counts, as long as they are not allocated in managed memory. The OpenARC runtime processes data clauses left-to-right. However, in Fig. 2, because memory transfers are always performed, the runtime will copy the data in at entry and copy it out at the exit.

4.6. Reusing device memory

Memory allocate and free can be expensive operations. Some runtimes try to improve performance by managing free memory pools internally.

4.6.1. PGI

As with OpenARC (we will describe in the next section), the PGI runtime uses a free device memory pool for CUDA devices.

4.6.2. OpenARC

The OpenARC implementation uses two memory pools to improve memory management. The first is a pool of free device memory. When device memory is freed, it is placed in the free memory pool. When a new device memory is allocated, it reuses free memory from the pool if a block of the same size exists. The memory in the pool is all deallocated if a device allocation runs out of memory. The second technique keeps track of both pinned host memory and the corresponding device memory when the device memory is freed. If the same host data is moved to the device, the saved device memory is reused, saving both host data pinning and device memory allocation costs. However, this second technique may suffer from too much memory pinning, causing slowdown or even program faults. Therefore, OpenARC uses the first technique by default, and the second one is optional for advanced users.

4.7. Partially overlapping data and adjacent data

On a *classic accelerator* with separate memory spaces, a problem can arise when the program moves a block of data to the device, then later wants to move a larger or intersecting block of data. To handle this case properly, the two overlapping data should be merged either by extending the device memory allocated for the old data or by allocating the union of all of the old and new data and re-initializing the memory. However, for either scenario, handling this properly is very challenging, so all of the compilers described in this article issue a runtime error in this case. A related problem is moving two contiguous blocks of data separately. For the same reasons as the partially overlapping data case, the memory for the second block of data cannot be guaranteed to be contiguous with the first block of data. For this reason, trying to refer to the whole array is not allowed in OpenACC. In the event the whole array is referenced, the PGI implementation will process the data directives in sequence but will generate bad data references in the parallel loop. Both the OpenARC and OpenUH implementations, on the other hand, would produce a runtime error message when handling the data directive for the second block.

5. Deep copy

The OpenACC committee continues to work on additional features, particularly on supporting *deep copy* of nested data structures [18,19]. For instance, a dynamically allocated array of a structured data type (C *struct* or Fortran *derived type*), the data type has one or more members that are themselves dynamically allocated arrays, such as structured data types with additional allocatable members. Some of these are conceptually simple (the C++ *vector* class), but describing how to copy the data to the device and back becomes deceptively complex.

A simple example is shown in Fig. 3. In this example, the programmer expects that copying the *X* struct will copy the data to its member points as well. The first problem is to tell the runtime how much data starting at *X.d* should be copied. After copying that data, the pointer, *d*, on the device must be replaced with the address of the new data. Any time the struct *X* gets copied back to the host, that device pointer address must not be moved back to the host. Deallocating or reallocating the data pointer on the host or on the device must either be defined as invalid, or the runtime must replicate the behavior in the other memory.

```
typedef struct{
    float *d; size_t n; float coef;
} vtype; vtype X;
#pragma acc enter data copyin(X)
```

Fig. 3. Simple deep copy.

```
typedef struct A { float* x, *y; ...  }
A a; a.x = malloc(N*sizeof(float)); a.y = a.x;
#pramga acc enter data copyin(a)
#pramga acc enter data copyin(a.x[0:N])
#pramga acc enter data copyin(a.y[0:N])
...
#pramga acc exit data copyout(a.x[0:N])
#pramga acc exit data copyout(a.y[0:N])
#pramga acc exit data copyout(a)
```

Fig. 4. Attach and detach example in C.

Another challenge from an OpenACC application is to handle large struct arrays with dynamic members. The ICON climate code [20] developed by the German Weather Service and the Max Planck Institute for Meteorology uses nested derived type arrays. Each element of the main state array has many member arrays, some of which are derived types with more member arrays. Even though the entire structure is quite large, only a subset of the entire structure is needed on the device during each phase of the computation. Moreover, some of the members are only read during some of the phases. In the interests of performance, we want the ability to optimize how much data is allocated on and copied to the device, as well as how much is copied back to the host. This requires being able to select whether to allocate only some members, and whether to copy some members in to device memory and other members out to host memory, and to allow different selections in different phases.

There are two possible directions to solve this problem. One direction is to require the programmer to provide *device constructor* and *destructor* routines, like the C++ object constructors and destructors. These routines would perform data allocation in device memory and movement between host and device memories. The OpenACC runtime would only have to know which routines to invoke at which points in the program. Although feasible, this is a heavy burden for the programmer. A second direction is to specify the various options using directives in the data type declaration. There is a small set of required behaviors that users want, such as selecting which members to copy and in which direction, specifying the size of shape of the data, and allowing different behavior for objects of the same data type in different points in the program.

In terms of the deep copy support in compilers, OpenACC compilers that support full deep copy and manual deep copy do exist. Full deep copy is intended to copy over the entire structure of nested elements for an aggregate data variable. Both Cray and PGI have implemented full deep copy support for Fortran applications. Fortran dynamic data structures including pointers and allocatable arrays contain self-describing dope vectors, each of which holds dimensional information and base addresses of every single dynamic data structure. Programmers do not have to insert additional directives to handle complex aggregate data. The compiler traverses the entire nested complex data structure and generates dynamic structure descriptors for the runtime to track data movement. This functionality significantly reduces workload of the ICON application porting in terms of manually managing every single elements movement between the host and device memory. However, some members of a structured data variable may not be used on the device, and moving these data may cause overhead.

The recently (November 2017) ratified specification, OpenACC 2.6, defines manual deep copy behaviors with attach and detach operations. Both attach and detach operations are pointer address setup in device memory. These operations are under the programmer's control, indicating that the programmer can manually manage the necessary data movement.

Consider the example in Fig. 4, the device copy of *a* has been allocated in device memory, but its members still point to the host memory location. In a non-shared memory system, any access to this memory location from the device is likely to cause the application to crash. Before data is accessed, pointer members on the device memory have to be set to their respective memory on the device. This action is called *attach*. Now pointer members direct the device memory after the attach operations. In this example, clause *copyin(a.x[0:N])* allocates the memory of *x* and also performs the attach action for *a.x*. Now if variable *a* is copied back to the host, the pointer member *x* in the host memory points to a device memory location, and it may cause a runtime error when the host tries to access it. Before the data is copied back to the host, the pointer member has to be reset to the host's memory address. This procedure is called *detach*. Basically, attach and detach operations are small data transfers between the host and device. Frequently attach and detach operations may cause significant overhead.

Minimizing data traffic of attach and detach operations becomes an optimization topic. A simple way to reduce the attach and detach is to check the present counter. If the present counter turns from 0 to 1, the attach operation is performed with pointer address being set in device memory. If the present counter becomes 0, the detach operation is performed by resetting the respective address in the host address. However, this method cannot handle multiple pointers to the same

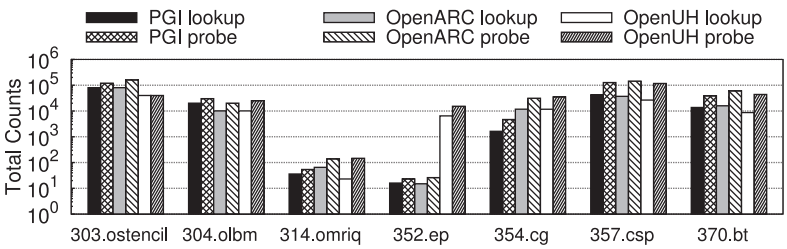

Fig. 5. Present table lookup and probe.

device memory. Consider the example in Fig. 4, *a.y* is not set to device pointer address when *copyin(a.y[0:N])* is executed with reference counter becoming two. When the exit directives are executed, the pointer member *x* still points to the device memory address. In terms of deep copy, this is an incorrect behavior.

In a PGI compiler, an attach table is used to track the attachment counter and minimize attach and detach data transfer. An attach table is implemented as a red–black tree with the address of host pointer as the search key. An attachment counter is associated with each search key. If the address of a pointer member is not found in the attach table, an attach operation happens. If attachment counter becomes 0, a detach operation is performed. Ideally, attach and detach data transfer can be reduced to one occurrence for each pointer member if their targeted address is fixed.

Although these are still directive-based approaches, they can be quite cumbersome for programmers to exploit for their applications. As a result, the OpenACC technical committee is currently working on an improvised directive-based approach to specify data movement in the data structure declaration. Essentially, this will allow users to explicitly shape member pointers, which will let C/C++ pointers be self-describing. It will be critical to design the directive syntax to be functional as needed and as natural as possible.

6. Evaluation

This section evaluates the different implementations of the OpenACC data model chosen by the three OpenACC compilers (the PGI OpenACC compiler, OpenARC, and OpenUH). Because it is difficult to quantitatively measure the performance behavior of each feature of the OpenACC data model independently, we only measured four representative features: *present table lookups, device memory allocation, pinned memory allocation*, and *managed memory*. The examined features cannot be captured using the standard OpenACC profiling interface, so we modified the built-in profiling tools of each compiler to measure these features. We evaluated seven OpenACC applications from the SPEC ACCEL benchmark suite V1.1 [21] on an NVIDIA GPU. (The evaluated features are not affected by the target device.) For more detailed analysis on a real application, we also evaluated a shock-hydrodynamics mini-application called LULESH [22]. For input data, we used reference data sets for the SPEC ACCEL benchmark suite and the default domain size (30^3) for LULESH, respectively.

6.1. Present table lookups

Fig. 5 shows the number of present table lookups, where *lookup* refers to the number of the present table searches requested by a generated OpenACC code, and *probe* refers to the number of present table entries actually inspected by the underlying implementation. The figure shows several interesting findings; first, each implementation requests a different number of present table lookups for the same program, and OpenUH requests the least numbers, except for *352.ep* and *354.cg*. The number of lookups will depends on how the compiler translates data constructs; as shown in Section 4.1.3, OpenUH uses a static hash table, which helps the compiler save the device address of the data once for all compute regions within structured data constructs, reducing the number of present table lookups. In *352.ep* and *354.cg*, compute regions are enclosed by an outer loop. If the device addresses of the data accessed in the enclosed compute regions are looked up at the corresponding kernel invocation sites, the present table lookups will be repeated at every iteration of the enclosing loop. PGI reduces these recurring lookups by hoisting the lookups out of the enclosing loops, resulting in the least lookups for both *352.ep* and *354.cg*. However, OpenARC applies the hoisting only to *352.ep*, and OpenUH does not apply it.

The ratio of the probe count to the lookup count in the figure indicates that the red–black trees in PGI and OpenARC work reasonably well on the tested benchmarks. The ratio in OpenUH has a higher variance than PGI and OpenARC, which is because the present table in OpenUH uses a hash table. A hash table can find mapping in $O(1)$ time if the key exists in the mapping, but if the key represents a partial data range, the entire hash map may be checked to determine if the partial data range is within the whole data range.

6.2. Device memory allocation

Fig. 6 shows the number of device memory allocations, where *create* refers to the number of requests that an OpenACC program requests to allocate data on the device, and *malloc* indicates the actual number of device memory allocations by

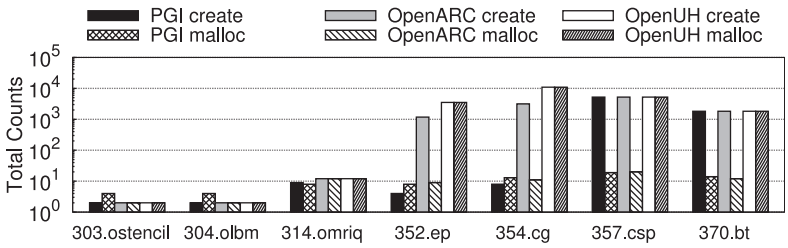

Fig. 6. Device memory create and malloc.

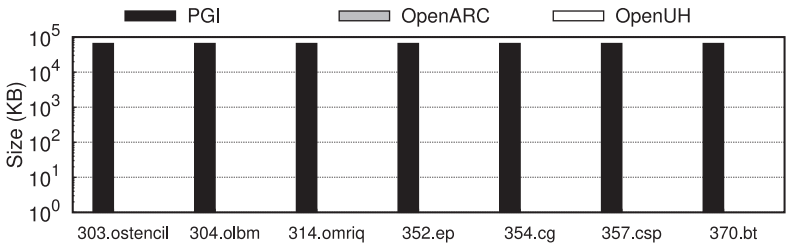

Fig. 7. Pinned memory size (KB).

the underlying implementation. This figure also reveals several interesting results: First, in the PGI implementation, the malloc count can be more than the create count, which is because PGI allocates additional runtime buffers. Second, in some benchmarks, PGI and OpenARC have much smaller malloc count than the create count, which is because both PGI and OpenARC reuse device memory by managing free memory pools internally (Section 4.6). Third, in *352.ep* and *354.cg*, three implementations have noticeable differences in both the create and malloc counts. As mentioned in the previous section, these benchmarks have compute regions enclosed by an outer loop, and thus memory allocation hoisting optimization plays a critical role in reducing the overall device memory allocations.

6.3. Pinned memory allocation

As explained in Section 4.2, an asynchronous data transfer requires that the host memory is pinned, not to be paged out by the operating system, and multiple ways to support asynchronous data transfers using the pinned memory exist. Fig. 7 shows the amount memory explicitly pinned by the OpenACC runtime of each implementation, where the PGI implementation uses the same amount of pinned memory (64 MB) for all applications, whereas the OpenARC and OpenUH implementations do not explicitly allocate any pinned memory. This is because both OpenARC and OpenUH use a *lazy pinning* approach, and PGI uses pinned buffers for memory transfers. Lazy pinning methods in OpenARC and OpenUH pin the host data only if they are transferred asynchronously, and the seven tested benchmarks do not have any asynchronous transfers; therefore, both OpenARC and OpenUH do not incur any explicit memory pinning. However, PGI always uses fixed-sized pinned buffers internally to overlap the data copy to the pinned buffer with the transfer between the buffer and the GPU (i.e., double buffering).

6.4. Managed memory

As explained in Section 4.1, both PGI and OpenARC experimentally support *managed memory* such as the CUDA unified memory. The PGI implementation offers a command line flag, which automatically changes the input program to the version that allocates user data on the CUDA-managed memory by replacing memory allocation calls. On the other hand, OpenARC requires a user to manually replace host memory allocation calls with OpenARC-specific memory management API calls. Once converted either automatically (PGI) or manually (OpenARC), the input program can seamlessly run on a system with the managed memory by letting the underlying driver manage the data. Fig. 8 compares the amount of explicit memory transfers by the OpenACC runtime, when the tested benchmarks are executed in a managed memory mode and a discrete memory mode. The figure shows that not all user data are implicitly managed by the driver even in the managed memory mode. In PGI, the automatic conversion is not applied to the statically allocated data (e.g., *352.ep, 354.cg, 357.csp*, and *370.bt*). In OpenARC, the user can convert both static and dynamic data, but temporary data, such as those used for reduction operations that are automatically created by the implementation, do not use the managed memory (e.g., *352.ep, 354.cg*, and *357.csp*). Therefore, some user data are explicitly transferred by the OpenACC runtime even in the managed memory mode. Nonetheless, supporting managed memory is very useful because it allows the same OpenACC program to run on multiple different memory architectures (Fig. 1) with minimal changes, maximizing its portability.

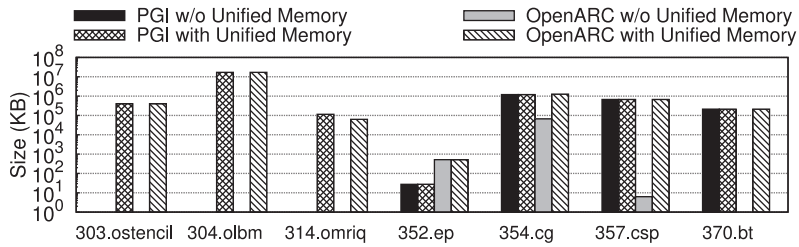

Fig. 8. Explicit memory transfer size (KB).

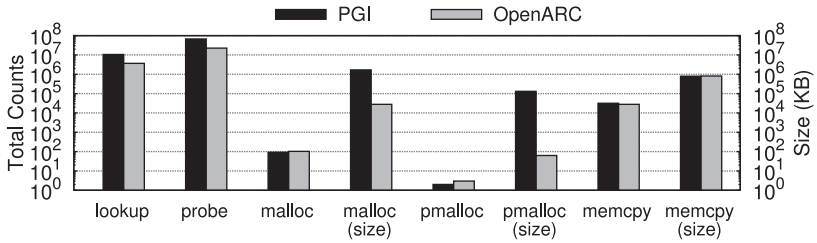

Fig. 9. OpenACC data model behaviors on LULESH. The left vertical axis corresponds to the numbers of lookup, probe, malloc, pmalloc, and memcpy calls. The right vertical axis corresponds to malloc, pmalloc, and memcpy data sizes. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.).

6.5. Detailed analysis using LULESH

For more detailed analysis of the different OpenACC data model implementations on a real application, we evaluated LULESH, a shock-hydrodynamics mini-application implemented by Lawrence Livermore National Laboratory as one of five challenge problems in the DARPA UHPC program. Fig. 9 shows the detailed information on the present table lookups (lookups and probes), device memory allocation (malloc count and size), pinned memory allocation (pmalloc count and size), and host-device memory transfers (memcpy count and size) behaviors, when running LULESH. The similar ratio of the probe count to the lookup count in the figure shows that the red–black tree mechanism works well on the real application, as well as the benchmarks, for both PGI and OpenARC (Fig. 5). The information on the device memory allocation and the pinned memory allocation reveals that even though both PGI and OpenARC generate output programs that execute the similar amount of memory allocation calls, the allocated memory sizes differ. This is because PGI uses a set of fixed-sized, additional buffers for internal memory management. Both PGI and OpenARC reuse device memory by managing free memory pools internally, but the figure may suggest that OpenARC exploits the free device memory pool more aggressively than PGI in the tested application. The figure also indicates that both PGI and OpenARC implementations have similar memory transfer behaviors, which is expected because memory transfer behaviors are usually dominated by the OpenACC data clauses inserted by users. If user data that are accessed in a compute region but not included in any of enclosing data clauses exist, the OpenACC compiler should decide the appropriate memory transfer behaviors; in LULESH, most user data are explicitly included in data clauses, and for implicitly determined data, both PGI and OpenARC handle them similarly.

7. Summary and future work

We described how several current implementations support the OpenACC data model, highlighting the different choices made. This includes implementation of the data directives and clauses, testing whether the data is already present on the device, managing asynchronous data transfers and memory allocations, handling aliased data, reusing device memory, managing partially present data, and supporting shared memory between the host and device. The goal is to provide information and guidance for other implementations of OpenACC or similar programming models such as OpenMP. State-of-the-art devices have a more interesting variety of memory hierarchies [23–25]. The OpenACC specification and implementations must evolve to support many different memory hierarchy organizations to provide a truly performance-portable experience.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research. This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.parco.2018.07.003

References

[1] R.W. Hockney, C.R. Jesshope, The FPS AP-120b, FPS164 (M140, M30), 264 (M60), 164/MAX (M145), CRC Press, 1988, pp. pp.206–243.
[2] P.M. Kogge, The IBM 3838 Array Processor, CRC Press, 1981, pp. pp.164–166.
[3] J. Fang, H. Sips, L. Zhang, C. Xu, Y. Che, A.L. Varbanescu, Test-driving Intel Xeon Phi, in: Proceedings of the Fifth ACM/SPEC International Conference on Performance Engineering, ICPE '14, ACM, New York, NY, USA, 2014, pp. 137–148.
[4] D. Schneider, Could supercomputing turn to signal processors (again)?, 2012, http://spectrum.ieee.org/computing/hardware/could-supercomputing-turn-to-signal-processors-again/, accessed: 2018-01-20.
[5] The OpenACC Application Programming Interface, 2017, Version 2.6, accessed: 2018-07-21.
[6] T.D. Han, T.S. Abdelrahman, HiCUDA: high-level GPGPU programming, IEEE Trans. Parallel Distrib. Syst. 22 (1) (2011) 78–90. http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.62.
[7] S. Lee, R. Eigenmann, OpenMPC: extended OpenMP programming and tuning for GPUs, in: Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society, 2010, pp. 1–11.
[8] T.B. Jablin, P. Prabhu, J.A. Jablin, N.P. Johnson, S.R. Beard, D.I. August, Automatic CPU-GPU communication management and optimization, in: Proceedings of the Thirty-Second ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, ACM, 2011, pp. 142–151.
[9] S. Pai, R. Govindarajan, M.J. Thazhuthaveetil, Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme, in: Proceedings of the Twenty-First International Conference on Parallel Architectures and Compilation Techniques, PACT '12, ACM, 2012, pp. 33–42.
[10] PGI, PGI OpenACC compiler, http://www.pgroup.com/resources/accel.htm, accessed: 2018-01-20.
[11] J. Beyer, Use of OpenACC and OpenMP directives in CRAY compilation environment (CCE), http://on-demand.gputechconf.com/gtc/2013/presentations/S3084-OpenACC-OpenMP-Directives-CCE.pdf, accessed: 2018-01-20.
[12] R. Reyes, I. López-Rodríguez, J. Fumero, F. Sande, accULL: an OpenACC implementation with CUDA and OpenCL support, in: Proceedings of the Euro-Par Parallel Processing, in: Lecture Notes in Computer Science, vol. 7484, Springer Berlin Heidelberg, 2012, pp. 871–882, doi:10.1007/978-3-642-32820-6_86.
[13] S. Lee, J. Vetter, OpenARC: Extensible OpenACC compiler framework for directive-based accelerator programming study, in: Proceedings of the WACCPD: Workshop on Accelerator Programming Using Directives in Conjunction with SC'14, 2014.
[14] X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, B. Chapman, Compiling a high-level directive-based programming model for GPGPUs, in: Proceedings of the International Workshop on Languages and Compilers for Parallel Computing, Springer, 2013, pp. 105–120.
[15] M. Wolfe, S. Lee, J. Kim, X. Tian, R. Xu, S. Chandrasekaran, B. Chapman, Implementing the OpenACC data model, in: Proceedings of the Seventh International Workshop on Accelerators and Hybrid Exascale Systems (AsHES) in Conjunction with IPDPS17, AsHES '17, 2017.
[16] C. Dave, H. Bae, S.J. Min, S. Lee, R. Eigenmann, S. Midkiff, Cetus: a source-to-source compiler infrastructure for multicores, IEEE Comput. 42 (12) (2009) 36–42.
[17] B. Chapman, D. Eachempati, O. Hernandez, Experiences developing the OpenUH compiler and runtime infrastructure, Int. J. Parallel Program (2012) 1–30.
[18] Complex data management in OpenACC programs, http://www.openacc.org/sites/default/files/TR-14-1.pdf, accessed: 2018-01-20.
[19] Deep copy attach and detach, http://www.openacc.org/sites/default/files/TR-16-1.pdf, accessed: 2018-01-20.
[20] ICON, Icosahedral non-hydrostatic, http://www.dlr.de/pa/en/desktopdefault.aspx/tabid-8859/15306_read-41918/, accessed: 2018-01-20.
[21] Standard performance evaluation corporation, SPEC ACCEL benchmark V1.1, https://www.spec.org/accel/, accessed: 2018-01-20.
[22] Hydrodynamics Challenge Problem, Hydrodynamics challenge problem, Lawrence Livermore National Laboratory, Technical Report LLNL-TR-490254.
[23] J.S. Vetter (Ed.), Contemporary High Performance Computing: From Petascale toward Exascale, CRC Computational Science Series, vol. 1, Taylor and Francis, Boca Raton, 2013.
[24] Intel knights landing yields big bang for the buck jump, https://www.nextplatform.com/2016/06/20/intel-knights-landing-yields-big-bang-buck-jump/, accessed: 2018-01-20.
[25] NVlink takes GPU acceleration to the next level, https://www.nextplatform.com/2016/05/04/nvlink-takes-gpu-acceleration-next-level/, accessed: 2018-01-20.