2017 IEEE International Parallel and Distributed Processing Symposium Workshops

Implementing the OpenACC Data Model

Michael Wolfe
The Portland Group
Email: mwolfe@nvidia.com

Seyong Lee and Jungwon Kim
Oak Ridge National Laboratory
Email: {lees2, kimj}@ornl.gov

Xiaonan Tian
The Portland Group
Email: datian@nvidia.com

Rengan Xu
Dell
Email: Rengan_Xu@dell.com

Sunita Chandrasekaran
University of Delaware
Email: schandra@udel.edu

Barbara Chapman
StonyBrook University
Email: barbara.chapman@stonybrook.edu

***Abstract*—Programming accelerators today usually requires managing separate virtual and physical memories, such as allocating space in and copying data between host and device memories. The OpenACC API provides data directives and clauses to control this behavior where it is required. This paper describes how the data model is supported in current OpenACC implementations, ranging from research compilers (OpenUH and OpenARC) to a commercial compiler (the PGI OpenACC compiler). This includes implementation of the data directives and clauses, testing whether the data is already present on the device, OpenCL support, managing asynchronous data transfers and memory allocations, handling aliased data, reusing device memory, managing partially present data, and support for shared memory between host and device. Lastly, it also discusses on-going work to manage large, complex dynamic data structures.**

I. Introduction

Compute accelerators have been used in high performance computing for many years. Early accelerators were attached array processors, such as from Floating Point Systems [1], IBM [2] and others, often programmed as a *subroutine box*. The main program would run on a minicomputer or mainframe and would call a subroutine that would be implemented on the array processor. The mechanics of moving input data to the array processor memory, invoking the array processor compute engine and bringing results back were hidden in the highly tuned subroutine. More recent compute accelerators include the Cell Processor [3] and the Clearspeed accelerator card [4], each programmed with language extensions.

Current compute accelerators include GPUs from NVIDIA and AMD, many-core coprocessors from Intel [5], digital signal processors (DSPs) from Texas Instruments [6] and field-programmable gate arrays (FPGAs). As each compute accelerator was introduced, a new programming language or methodology was promoted as well: CUDA [7] for NVIDIA GPUs, Brook+ [8] for AMD GPUs, a directive-based offload model [9] for Intel many-core coprocessors, low-level C and assembly programming for DSPs [10] as used for embedded applications, and hardware description languages for FPGAs.

High performance computing users demand a higher level programming model and portability across a range of architectures. One answer to this demand was the development of the OpenACC Application Programming Interface [11]. OpenACC uses directives, quite similar to the OpenMP API and various other directive sets [12], [13], 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 [13], [14], [15] 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.

As we look forward to the near and more distant future, we see a range of architectures being developed with more interesting memory architectures. OpenACC was initially designed to work with accelerator devices that have their own memory, as well as devices that share memory with the host. Section II describes a range of memory architectures that OpenACC is able to support. 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. Section III describes the OpenACC data model in more detail. There are now several commercial and research implementations of OpenACC [16], [17], [18], [19], [20]. Section IV describes how three current OpenACC compilers (PGI [16], OpenUH [20], and OpenARC [19]) implement the *present table*, support OpenCL devices, manage asynchronous data transfers, and more. Section V describes the work on adding *deep copy* behavior to OpenACC, which is needed to manage large, complex dynamic data structures. Section VI evaluates the present table and device memory management schemes of the tested three OpenACC compilers. Finally, Section VII summarizes the goals of this article.

978-1-5386-3408-0/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPSW.2017.85

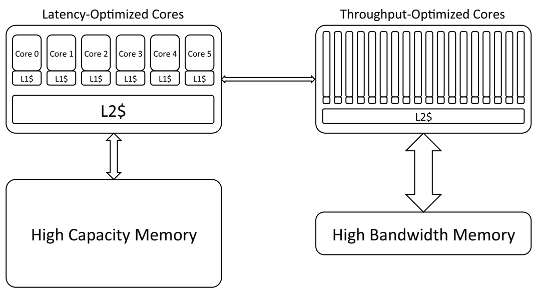


Figure 1: Classic accelerator model: separate physical and virtual memories.

II. Target Memory Architectures

This section describes several system memory architectures which OpenACC was designed to support. There are other significant design issues in these systems, but here we focus on the memories.

The *classic accelerator* memory architecture is shown in Figure 1. A latency-optimized multicore appears on the left, with a large coherent cache memory attached to a very high capacity memory (HCM) (often 64GB or larger). An accelerator composed of many throughput optimized cores appears on the right, typically with much smaller cache memory and attached to a much smaller (4GB-16GB) but much higher bandwidth memory (HBM). Today's accelerators do not use paging in the HBM, so the amount of data that can be mapped to the HBM is limited to its physical size. The typical connection between the host and accelerator today is through the PCI-express IO bus. To the multicore hardware and operating system, the accelerator is an IO device controlled by OS drivers. In the most limited case, the multicore 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

- allocate space in the HBM
- copy input data from the host memory to the HBM
- launch one or more procedures or kernels on the accelerator to do the calculations
- copy results from the HBM back to the host memory
- deallocate the space in the HBM

The datapath between the host and the accelerator has low bandwidth relative 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.

Some systems use the same physical layout, but can allocate memory in the HBM in such a way that it can be mapped into the host virtual address space. This allows a program to leave data in the HBM and allow both compute engines, the multicore and accelerator, to access it directly. There is a significant performance penalty for accesses from the multicore, however. Not only must the data be transferred over the relatively slow IO bus, it probably can't be cached in the multicore because the accelerator doesn't participate in the cache coherence protocol. This feature and the next are sometimes called *zero-copy* memory, because the program doesn't need to explicitly copy the data from one memory to the other. We call this feature *zero-copy device memory*. This feature is difficult to use in OpenACC, because the directive model doesn't control memory allocation.

Other systems use the same physical layout, but can allocate memory in the multicore memory such that it can be accessed directly from the accelerator. This memory must be allocated contiguously and *pinned* in the multicore physical memory, so that it won't get moved or replaced by the operating system paging mechanism. As above, there is a significant performance penalty for accesses from the accelerator, because of the low IO bus latency. We call this feature *zero-copy host memory*. This feature can potentially be used in OpenACC.

Current NVIDIA GPUs support a shared address space between the CPU and GPU, called *CUDA Unified Memory*. When a program allocates *managed memory* using special allocation routines, the runtime will allocate space in the CPU memory as well as the HBM. When a compute kernel is launched, the CUDA driver will move any managed memory data from the CPU memory to the HBM and mark the CPU pages as nonresident. This is necessary because the current GPUs have no page fault detection hardware. A subsequent CPU reference to managed memory space will trigger a page fault, 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.

Future NVIDIA GPUs will have the same physical layout as in Figure 1, but with 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. In particular, it will allow the accelerator to access data in host memory without being specially allocated, and without pinning the data, though 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, as we shall see. We call this design *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 (APU). In this design, both latency-optimized cores and throughput-optimized

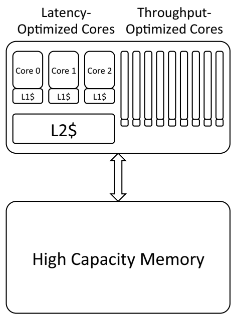


Figure 2: Single physical memory model: latency-optimized and throughput-optimized cores on one processor.

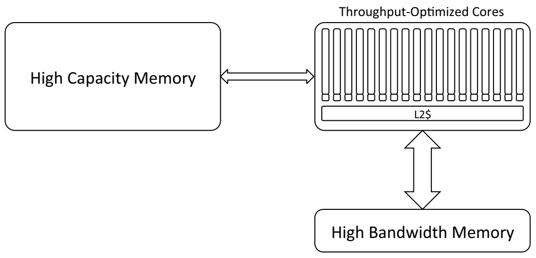


Figure 3: Throughput only model: throughput-optimized cores with both HCM and HBM.

cores are included on a single processor chip, both using the same cache hierarchy and the same memory interface to the host memory, as shown in Figure 2. This design has several important advantages. Even if the operating system and software doesn't allow for 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 need not 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 Figure 3. 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 afterwards. 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 with no HBM and no 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 *multicore*.

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, say 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.

In some cases, the decision is easy. For the *multicore* and *single physical memory* targets, there is no HBM and no data movement. For the *classic accelerator* target, data must be copied and the only decision is when. For other targets, some or all data may be allocated and remain in host memory, or may be allocated and remain in the HBM, or may be copied or moved between them. In the *throughput-only* or *unified virtual memory* targets, the system could use paging hardware and operating system or driver software to decide at runtime whether to move pages of data from one memory to another, or could require the application or runtime to explicitly move or copy data between the memories.

III. OpenACC Data Model

The OpenACC execution model assumes that the program starts execution on a *host* with one or more attached *accelerators*. OpenACC has data clauses to tell the implementation when to copy data from the host memory to the device memory, assuming one exists. 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*, 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 very like the entry to a structured *data* construct, and the *exit data* directive acts very like the exit from a structured *data* construct.

```
#pragma acc data copy(a[0:n],b[0:n])
{
   for( int i = 0; i < n; ++i ) {
       a[i] = ...
   }
}
```

Figure 4: Structured *data* constructs.

For some targets, the data directives can be completely ignored. For a *multicore* 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, since the accelerator cores cannot access host memory. For *zero-copy* memory (either type) or *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, local to a procedure (on the stack), or 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. We haven't seen these systems yet, and one could imagine an operating system module to detect a high number of accesses to the main memory, and to decide to move the data to the HBM without any input from the application or runtime. Nevertheless, we expect to use the OpenACC data clauses to prefetch data to the HBM. However, currently there is no way to distinguish between a data clause inserted for correctness on a *classic accelerator* target and one inserted for performance tuning on a *unified virtual memory* target.

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, while 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.

IV. The Implementations

This article describes how the OpenACC data directives are implemented in three compilers. OpenUH [21], [20] 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.

OpenARC [19] is an open source, extensible research compiler framework based on Cetus [22], 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 FPGAs. See ft.ornl.gov/research/openarc for more information about OpenARC.

PGI provides commercial compilers for C++, C and Fortran which include the OpenACC 2.0 directives, OpenMP 3.1 directives, and many other features [16]. The OpenACC directives grew out of the PGI Accelerator directives which were first introduced in 2008. See www.pgroup.com for more information about the PGI compilers.

A. 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. Variables names can't be used since they don't persist when pointers are passed as arguments to a routine, and an implementation can't add information to a pointer (*fat pointer*) since a user doesn't 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. At a minimum, the present table will return the corresponding device address for host data. Table lookup should be fast in the common case.

1) PGI: The PGI OpenACC runtime uses a Red-Black tree indexed by the host address range for the *present table*. Since C and C++ programs often do pointer arithmetic and Fortran programs often pass subarrays as arguments to other routines, this implementation can accept any address in the range of the data object and return the appropriate device address. A Red-Black tree is a reasonably balanced binary tree, so lookup, insert and delete operations take $O(\log n)$ time. The table entry stores the start and end address of the corresponding host data, the start address of the device data, and the two reference counts. It may store additional information to help with data inspection, such as data type and filename or function name and line number where the data was first created in device memory. 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.

2) OpenUH: The OpenUH compiler uses hash tables for the present table implementation. It maintains two hash tables: static and dynamic 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 the 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. Figure 5a shows an example OpenACC code and Figure 5b shows the structure of the region stack. 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 region level i 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 (for `copy` and `copyin` clauses). 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.

3) OpenARC: The OpenARC runtime implements the *present table* using a C++ Standard Template Library (STL) 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 [19]. These routines allocate and free managed memory, and can fall back to using system memory if the target device does not support managed memory. The present table also keeps track of data allocated in managed memory by using the host address for both the table key and the 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, but which also execute correctly on systems which do not support managed memory.

B. OpenCL

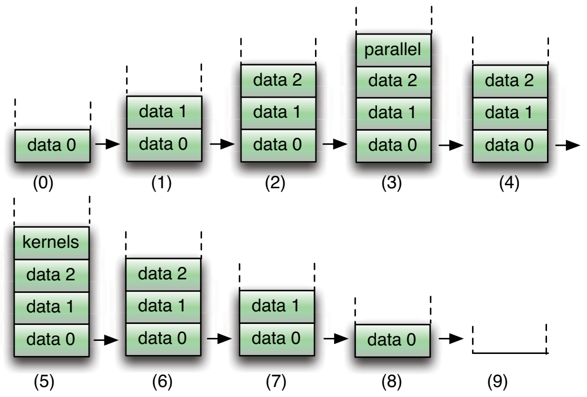
Data management with CUDA is very like data management in C. A memory allocation routine returns an address; the program can add offsets to this address; memory deallocation is done by passing the allocation address to a *free* routine. This is convenient, since the host program can capture the device address of an array, and perform address arithmetic on the host to produce another valid device address.

The OpenCL interface works quite differently. With OpenCL, the memory allocation routine returns a buffer handle, that is, the address to an opaque struct that the OpenCL runtime uses to describe the memory buffer. No address arithmetic can be performed on the handle. The OpenCL buffer may or may not be resident in device memory for the whole time between the allocate and free operations. When the buffer is moved to the device, there is no guarantee that it will be at the same memory address each time. This makes it difficult to provide the same capabilities as with the NVIDIA CUDA interface.

1) OpenARC: The OpenARC compiler supports OpenCL on NVIDIA/AMD GPUs, Intel Xeon Phi Coprocessors, and Altera FPGAs. The OpenARC implementation uses a fake virtual device address space; the present table for an OpenCL device maps a host address to a tuple with a fake device pointer, an OpenCL handle, and the data size. Because the fake device address can not be used in any user-written OpenCL kernels, the runtime also provides an API routine that returns the OpenCL handle and offset corresponding to the fake device address. The current OpenARC runtime implements the virtual device address space using the CPU malloc() calls; when device memory is allocated, the OpenARC runtime also allocates dummy space in the CPU memory space, and uses the allocated address as a fake device address. The fake virtual device address is used

```
#pragma acc data create(...) //data 0
{ //(0)
  #pragma acc data create(...) //data 1
  { //(1)
    #pragma acc data create(...) //data 2
    { //(2)
      #pragma acc parallel create(...)
      { //(3)
        ...
      } //(4)
      #pragma acc kernels create(...)
      { //(5)
        ...
      } //(6)
    } //(7)
  } //(8)
} //(9)
```

(a) OpenACC code example



(b) Region stack flow for code in (a)

Figure 5: Runtime region stack structure.

only for the CPU-GPU address mapping and device address calculations, and no data are stored in the allocated dummy memory. Therefore, using the dummy memory consumes only virtual CPU memory space, but no physical CPU memory.

2) PGI: The PGI compilers support AMD GPUs and APUs using their OpenCL 1.2 toolkit and libraries. The PGI implementation uses a behavior specific to the AMD OpenCL library. That behavior guarantees that a buffer will always be placed at the same address on the device. The PGI runtime allocates the largest buffers supported on the AMD device, then implements its own memory manager within these *bigbuffers*. When a bigbuffer is allocated, the runtime launches a kernel that returns the device address where it starts. To ensure that the buffers are resident on the device, each OpenCL kernel launch has three (or more) arguments, which are handles to the global bigbuffers. Currently, the PGI compilers do not take advantage of the single physical memory for AMD APUs.

3) OpenUH: OpenUH compiler adopts two different approaches to support AMD GPUs. Both the approaches generate OpenCL kernel functions for the GPUs. The difference in the approaches lies on the CPU side. In the first approach, for detached non-unified memory AMD GPUs, OpenUH compiler generates OpenCL runtime function calls to allocate and free data memory. As the aforementioned limitation of OpenCL memory handler, the solution we proposed for CUDA cannot be applied to OpenCL to handle the subarray problem. The subarray will be allocated within a new OpenCL memory handler. In the second approach, for the shared memory architecture, such as heterogeneous system architecture (HSA)-based [23] AMD APUs, OpenUH backend generates the host code binary built on top of HSA runtime in order to share the memory pointer between the CPU and the GPU. In this approach, there is no concept of present table.

C. 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 be *pinned*, that is, allocated or moved to contiguous physical memory that the operating system will not page. This allows the DMA 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.

1) 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, since 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 find the right time for pinning.

For OpenCL devices, the OpenARC runtime relies on the OpenCL nonblocking data transfer features.

2) PGI: The default PGI implementation for NVIDIA GPUs is to use two pinned buffers for upload and for download, to allow 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.

The PGI OpenACC implementation has close interfaces to PGI CUDA Fortran. If an CUDA Fortran allocatable array has the *pinned* attribute, the compiler passes this information to the OpenACC runtime and true asynchronous data transfers are generated.

As with OpenARC, the PGI runtime just sets the non-blocking flag for asynchronous transfers when targeting OpenCL devices.

3) OpenUH: In the OpenUH implementation, the structure of the data not only includes the host address, device address and data size, but 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.

D. 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 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 also necessarily synchronous.

1) 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 no additional copy between the internal buffer and the host data is needed. 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.

2) 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 (FIFO) task queue which 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. Memory allocation will behave synchronously however, with the main thread waiting for the helper thread to provide the device address.

3) 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 the device memory doesn't get freed until all transfers are complete.

One could imagine an implementation in which, at an asynchronous allocate, the runtime searches through the *async* queue for an asynchronous free, and reuse that address for this 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 looking for pending memory free operations.

E. Aliasing on a Data Directive

When there are two or more objects specified on a data directive that are aliased with each other, OpenACC 2.0 is silent on the intended behavior. In Figure 6, the programmer

```
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 );
```

Figure 6: Data aliasing example.

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.

1) PGI: In the PGI implementation, the data clauses are processed left-to-right at the top of a data construct. In Figure 6, 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 since it is a *copyin* clause, will simply free the array.

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 Figure 6, because memory transfers are always performed, the runtime will copy the data in at entry and copy it out at the exit.

F. Reusing Device Memory

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

1) 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 there exists a block of the same size. The memory in the pool is all deallocated if some device allocation runs out of memory. The second technique keeps a 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.

2) PGI: As with OpenARC, the PGI runtime uses a free device memory pool for CUDA devices. For OpenCL devices, the PGI runtime allocates a small number of *bigbuffers* and manages all user data as suballocations in those *bigbuffers*; see Section IV-B2.

G. Partially Present Data

On a *classic accelerator*, a problem can arise where the program moves a block of data to the device, then later wants to move a larger or intersecting block of data:

```
#pragma acc enter data create(a[0:n])
...
#pragma acc enter data create(a[0:2*n])
```

In this example, the second *enter data* directive wants to enlarge the size of `a` on the device. One possible way handle this is to actually extend the memory allocated for `a` on the device. However, when the original space for `a` was allocated, there was no way to know whether to reserve adjacent memory for future extension, so this is infeasible. Another possible way is to allocate space for the whole array `a` instead of a subset, but again the program won't necessarily know the extent of the whole array, particular in C or C++. A third way is to allocate the union of all the old and new data for the second directive, initializing this with the elements from the original array. This is a challenge because the program can actually capture the device address of an array, so moving the array makes this device address invalid. Also, these operations may occur while a procedure is simultaneously computing on the original data, so moving it would be invalid. OpenACC defines this behavior as a runtime error, and all the compilers described in this article issue a runtime error in this case.

V. Deep Copy

The OpenACC committee continues to work on additional features, in particular on support *deep copy* of nested data structures [24], [25]. For instance, an dynamically allocated array of a structured data type (C *struct* or Fortran *derived type*), the data type has one or more members which are themselves dynamically allocated arrays, perhaps themselves of structured data types with additional allocatable members, and so on. Some of these are conceptually simple (the C++ *vector* class) but become deceptively complex to describe how to copy the data to the device and back.

A simple example is shown in Figure 7. In this example, the programmer expects that copying the *X* struct will copy the data to which 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 that 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)
```

Figure 7: Simple deep copy.

Consider the C++ standard template *vector* class. A typical implementation is prototyped in Figure 8. The class has only three data members, pointers to the start of the data vector, a pointer to the first element past the end of the data vector, and a pointer just past the end of the allocated storage. The *begin* pointer points to actual data, but the other pointers must be handled differently. Not only do they not point to valid data, they may not even contain a valid address (that can be dereferenced). Copying the vector to device memory requires a way to tell the compiler and runtime how much data to allocate, to fill in the *begin* pointer with the address of that data, and to fill in the *enddata* and *endstorage* pointers with offsets from the *begin* value. To make this even more complicated, the data members are only accessible from within the class, but the *vector* class is typically implemented in a read-only system header file.

```
template<typename T>class vector{
    T *begin, *enddata, *endstorage;
  public:
    T* begin() { return begin; }
    T* end()   { return enddata; }
    size_t size() {return enddata-begin;}
    ...
};
```

Figure 8: C++ vector class implementation.

Another challenge from an OpenACC application is to handle large struct arrays with dynamic members. The ICON climate code [26] developed by the German Weather Service (DWD) and the Max Planck Institute for Meteorology (MPI-M) 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, show in part in Figure 9. The entire structure is quite large, and during each phase of the computation, only a subset of the entire structure is needed on the device. Moreover, some of the members are only read during some of the phases. In the interests of performance, we want to be able to optimize how much data is allocated on and copied to the device, and how much is copied back to the host. This requires being able to select whether to allocate some members and not others, 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. While feasible, we feel this is a heavy burden on the programmer.

```
type t_nh_state
type(t_nh_prog),allocatable::prog(:)
type(t_var_list),allocatable::prog_list(:)
...
end type
...
type(t_nt_state),allocatable::p_nh_state(:)
... p_nh_state(:)%diag%vt(:,:,:)
... p_nh_state(:)%prog(:)%vn(:,:,:)
```

Figure 9: ICON Fortran data structures.

A second direction is to specify the various options using directives in the datatype declaration. There is a small set of required behaviors that our users want, such as selecting which members to copy and in which direction, specifying the size of shape of the data (for C and C++, in particular), and allowing different behavior for objects of the same datatype in different points in the program. The OpenACC committee is working on designing directive syntax that is as functional as needed and as natural as possible.

VI. Evaluation

This section evaluates the different implementations of the OpenACC data model chosen by the three OpenACC compilers (the PGI OpenACC compiler, OpenUH, and OpenARC). Because it is difficult to quantitatively measure the performance behavior of each feature of the OpenACC data model independently, we measure only two representative features: *present table lookups* and *device memory allocation*. We evaluated 7 OpenACC applications from the SPEC ACCEL benchmark suite V1.1 [27] on an NVIDIA GPU. (The evaluated features are not affected by a target device.)

A. Present Table Lookups

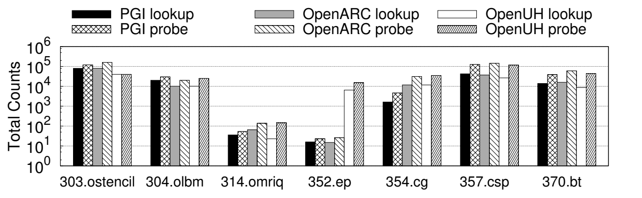


Figure 10: Present table lookup and probe.

Figure 10 shows the number of present table lookups, where *lookup* means 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 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 IV-A2, OpenUH uses a static hash table, which helps the compiler to 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 find whether the partial data range is within the whole data range.

B. Device Memory Allocation

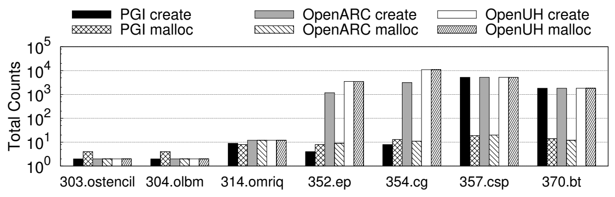


Figure 11: Device memory create and malloc.

Figure 11 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 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 the PGI implementation 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 IV-F). Third, in *352.ep* and *354.cg*, three implementations have noticeable differences in both the create counts and the 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.

VII. Summary

This article describes how several current implementations support the OpenACC data constructs, highlighting the different choices made. The goal is to provide information and guidance for other implementations of OpenACC or similar programming models. OpenMP 4.0 included new *target* extensions for attached devices. The OpenMP *target data* constructs are quite similar to the OpenACC *data* constructs and a runtime that supports one API can easily support both.

State-of-the-art devices have a more interesting variety of memory hierarchies [28], [29], [30]. Some devices have a simple flat memory, allowing an OpenACC implementation to ignore the data directives and clauses entirely. Some have both a large system memory and a separate high-bandwidth memory, providing the challenge and opportunity for an implementation to aggressively manage data traffic between the system memory and the high-bandwidth memory. And some have variations on the existing logically and physically separate device and system memories, requiring data movement between the two. The OpenACC specification and implementations must evolve to support many different memory hierarchy organizations in order to provide a truly performance-portable experience.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy.

References

[1] R. W. Hockney and C. R. Jesshope, *The FPS AP-120B, FPS164 (M140, M30), 264 (M60), 164/MAX (M145)*. CRC Press, 1988, pp. 206–243.

[2] P. M. Kogge, *The IBM 3838 Array Processor*. CRC Press, 1981, pp. 164–166.

[3] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, “The Potential of the Cell Processor for Scientific Computing,” in *Proc. of the 3rd Conference on Computing Frontiers*, ser. CF ’06. ACM, 2006, pp. 9–20.

[4] “CSX700 Floating Point Processor Datasheet,” https://www.electronicsdatasheets.com/download/5367711fe34e241020763e7c.pdf?format=pdf, accessed: 2017-01-20.

[5] J. Fang, H. Sips, L. Zhang, C. Xu, Y. Che, and A. L. Varbanescu, “Test-driving Intel Xeon Phi,” in *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE ’14. New York, NY, USA: ACM, 2014, pp. 137–148.

[6] D. Schneider, “Could Supercomputing Turn to Signal Processors (Again)?” http://spectrum.ieee.org/computing/hardware/could-supercomputing-turn-to-signal-processors-again/, 2012, accessed: 2017-01-20.

[7] J. Sanders and E. Kandrot, *CUDA by Example*. Addison Wesley, 2010.

[8] AMD, *Brook+ Programming*, 1st ed. Advanced Micro Devices, 2008, ch. 2.

[9] R. Farber, "Programming intel's xeon phi: A jumpstart introduction," *Dr. Dobb's*, Dec. 2012.

[10] *TMS320C6000 Programmer's Guide*, Texas Instruments, 2011, number: SPRU198K.

[11] *The OpenACC Application Programming Interface, Version 2.5*, http://www.openacc.org/sites/default/files/OpenACC_2pt5.pdf, 2015, accessed: 2017-01-20.

[12] 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.

[13] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP Programming and Tuning for GPUs," in *Proc. of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.

[14] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU communication management and optimization," in *Proc. of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '11. ACM, 2011, pp. 142–151.

[15] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil, "Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme," in *Proc. of the 21st international conference on Parallel architectures and compilation techniques*, ser. PACT '12. ACM, 2012, pp. 33–42.

[16] PGI, "PGI OpenACC Compiler," http://www.pgroup.com/resources/accel.htm, accessed: 2017-01-20.

[17] 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: 2017-01-20.

[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] S. Lee and J. Vetter, "OpenARC: Extensible OpenACC Compiler Framework for Directive-Based Accelerator Programming Study," in *WACCPD: Workshop on Accelerator Programming Using Directives in Conjunction with SC'14*, november 2014.

[20] X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman, "Compiling a high-level directive-based programming model for gpgpus," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2013, pp. 105–120.

[21] B. Chapman, D. Eachempati, and O. Hernandez, "Experiences Developing the OpenUH Compiler and Runtime Infrastructure," *International Journal of Parallel Programming*, pp. 1–30, 2012.

[22] 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.

[23] "Heterogeneous System Architectures," http://www.hsafoundation.com/, accessed: 2017-01-20.

[24] "Complex Data Management in OpenACC Programs," http://www.openacc.org/sites/default/files/TR-14-1.pdf, accessed: 2017-01-20.

[25] "Deep Copy Attach and Detach," http://www.openacc.org/sites/default/files/TR-16-1.pdf, accessed: 2017-01-20.

[26] ICON, "Icosahedral non-hydrostatic," http://www.dlr.de/pa/en/desktopdefault.aspx/tabid-8859/15306_read-41918/, accessed: 2017-01-20.

[27] Standard Performance Evaluation Corporation, "SPEC ACCEL benchmark V1.1," https://www.spec.org/accel/, accessed: 2017-01-20.

[28] 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.

[29] "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: 2017-01-20.

[30] "NVlink takes GPU acceleration to the next level," https://www.nextplatform.com/2016/05/04/nvlink-takes-gpu-acceleration-next-level/, accessed: 2017-01-20.