OpenACC and CUDA Unified Memory

Sebastien Deldon, James Beyer, Douglas Miles
PGI / NVIDIA

ABSTRACT: *CUDA Unified Memory (UM) simplifies application development for GPU-accelerated systems by presenting a single memory address space to both CPUs and GPUs. Data allocated in UM can be read or written through the same virtual address by code running on either a CPU or an NVIDIA GPU. OpenACC is a directive-based parallel programming model for both traditional shared-memory SMP systems and heterogeneous GPU-accelerated systems. It includes directives for managing data movement between levels of a memory hierarchy, particularly between host and device memory on GPU-accelerated systems. OpenACC data directives can be safely ignored or even omitted on a shared-memory system, allowing programmers to focus on exposing and expressing parallelism rather than on underlying system details. This paper describes an implementation of OpenACC built on top of CUDA Unified Memory that provides productivity benefits for porting and optimization of Fortran, C and C++ programs to GPU-accelerated Cray systems.*

KEYWORDS: Compiler, Accelerator, Multicore, GPU, Parallelization, Vectorization, OpenACC, Unified Memory, CUDA

1. Introduction

The OpenACC directive-based parallel programming model was designed to support migration of applications to GPU-accelerated and heterogeneous parallel HPC systems with several goals in mind:

- Higher programmer productivity compared to use of explicit models like CUDA and OpenCL
- Application source code instrumented with OpenACC directives should remain portable to any system with a standard Fortran/C/C++ compiler
- OpenACC applications should be performance portable across various types of parallel systems – multicore CPUs, heterogeneous CPU+GPU, and manycore processors

OpenACC enables development of code that is massively parallel and dynamically scalable at the node level to maximize performance on GPU-accelerated systems, and which can be readily compiled for parallel execution on multicore or manycore processors as well. One of the key features pioneered by OpenACC is directive-based data management, which allows the programmer to explicitly manage movement of data objects between host and accelerator device memory to support execution of parallel compute kernels on the accelerator [OACC15]. These directives were designed from the outset so that they could be safely ignored on a multicore CPU-based shared-memory system, enabling OpenACC parallel regions to be compiled for parallel execution on all cores of the system with no data movement overhead.

OpenACC has largely achieved the goals outlined above [HPCW17], and in the past year 3 of the 5 most widely-used HPC applications have adopted OpenACC;. Gaussian and ANSYS/Fluent have issued production releases using OpenACC for GPU acceleration, and the developers of VASP have disclosed plans to do so. Experience with these codes and many other applications, mini-apps and benchmarks has shown that explicit management of data between host and device memory is one of the most time-consuming and error prone aspects of OpenACC programming [GTC18]. While the resulting code is more portable, elegant and readable than comparable CUDA or OpenCL, the data management task remains a daunting aspect of porting large production applications using OpenACC, especially those that make extensive use of aggregate data types.

Over the past few years, PGI has added support for OpenACC targeting NVIDIA GPUs leveraging CUDA Unified Memory [UM13]. When compiling an OpenACC program in this mode, the PGI Fortran, C and C++ compilers intercept all visible and compiler-generated malloc/free, new/delete and allocate/deallocate calls or statements and place the resulting allocatables in CUDA Unified Memory. Implicitly allocated data, for example Fortran automatic arrays, are also placed by default in Unified Memory.

The benefit of compiling in this mode is that programmer-directed OpenACC data management is no longer necessary for allocatable data. The effect is to allow programmers to port and write GPU-accelerated OpenACC programs focusing only on parallelism, allowing the CUDA Unified Memory manager to handle most data movement implicitly. Data directives are still required for global, static or stack data, but for many modern applications this is minimal.

Using OpenACC with CUDA Unified Memory on standard benchmarks and applications has generally delivered good performance relative to user-managed data movement, as demonstrated by results in the sections below. The sections that follow give an overview of the implementation, use, results, limitations and future directions for support of OpenACC and CUDA Unified Memory.

2. Implementation

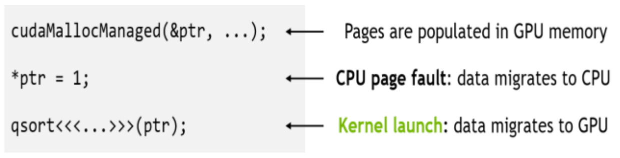
2.1 CUDA Unified Memory

Unified Memory support was added to the CUDA programming model in the CUDA 6.0 Toolkit. At that time, it targeted the Kepler GPU architecture using a pure software approach. Data allocated in Unified Memory using the cudaMallocManaged API was migrated en masse to device memory upon kernel launch, and any CPU access of that data would trigger a data transfer from the device back to host memory [GTC16].

While useful for prototyping and development, this scheme had several limitations when running on an NVIDIA Kepler GPU. The aggregate size of data in Unified Memory was limited to the GPU device physical memory size, and any attempt at simultaneous access to unified data from the CPU while in use by a GPU kernel caused programs to abort with a segmentation fault. In addition, without hardware support, the pure software implementation on Kepler could result in significant overheads in some cases.

The NVIDIA Pascal P100 GPU architecture improved Unified Memory functionality with the addition of 49-bit virtual addressing and use of a dedicated hardware page migration engine [NV16]. Unified Memory size on Pascal and the latest Volta V100 GPUs is not limited to GPU physical memory size, and concurrent access to data in Unified Memory between CPU and GPU is fully supported.

As noted earlier, data is allocated in Unified Memory using the cudaMallocManaged API call. An address returned by this API call can be referenced either by CPU Code or GPU kernels. A touch of 'managed' data triggers data migration if data is not available in the requested memory.



```
cudaMallocManaged(&ptr, ...);
*ptr = 1;
qsort<<<...>>>(ptr);
```

Pages are populated in GPU memory
CPU page fault: data migrates to CPU
Kernel launch: data migrates to GPU

Migration of data between CPU and GPU memories involves data transfer over PCI Express or NVLINK, and occurs in blocks that may be a single page or multiple pages. The algorithms for how much data to migrate and when to migrate it from memory to memory are being continually tuned in each CUDA release. These transfers are relatively slow compared to direct accesses to local memory, so optimizing data placement and movement is critical to achieving good performance. Also, keeping the CPU/GPU page table current is time consuming; updating the page table after a page fault can take tens of µs per page, potentially stalling execution.

Some programs may incur a performance penalty when using CUDA Unified Memory versus use of explicit OpenACC directive-based data transfers. The latter are often carefully placed to ensure data is present in device memory before it's needed, and to minimize the number of times data moves between memories. The API calls cudaMemAdvise and cudaMemPrefetchAsync have been introduced to enable the programmer to give hints to the Unified Memory manager and improve UM performance.

cudaMemAdvise enables specifying placement and access policies on regions of managed data. For example, when data is known to be read-only on the GPU, a 'read mostly' policy can be set for that data; those pages are then duplicated in GPU memory without being evicted from the CPU page table, thus avoiding needless data migration. Setting a 'preferred location' policy for data in Unified Memory can minimize page migrations for which the overhead exceeds the locality benefit, minimizing page thrashing. cudaMemPrefetchAsync prefetches an area of managed memory and can lower the cost of data movement and page table updates by overlapping these with either CPU or GPU-side computation [NV18].

2.2 Data Management in OpenACC

OpenACC is designed to support migration of applications to GPU-accelerated and heterogeneous parallel HPC systems. The model assumes that serial code is executed by a fast host CPU core and parallel code is executed on an accelerator. The model doesn't make any assumption of the accelerator type, which could be a discrete accelerator or the host itself operating in parallel or accelerator mode. The model allows for accelerator devices with a separate memory sub-system, but doesn't assume separated memories between host and accelerator. In this context, two main types of directives are defined by the OpenACC programing model: Compute Directives and Data Directives.

Compute directives are used to expose parallelism, to specify regions of code suitable for an accelerator, and to provide the compiler with hints on how parallelism can be efficiently mapped to a given accelerator. The two main compute directives are `parallel` and `kernels`. The `parallel` directive initiates parallel execution on an accelerator, and typically marks a region that includes one or more loops annotated with parallel loop directives. The `kernels` directive denotes a region of code the compiler should process by searching for parallelizable loops, either through auto-parallelization and auto-offloading techniques, or through parallelization and offloading of parallel loops explicitly annotated with OpenACC loop directives. Each loop nest in a `kernels` region is typically launched as an individual GPU kernel, whereas a `parallel` region results in a single kernel launch on a GPU accelerator.

OpenACC data directives are used to specify and optimize data transfers between host and accelerator to guarantee correct execution when their memories are separated. A data region can be either structured, meaning it starts and ends in the same syntactic block of code, or unstructured, starting in a given routine (C++ constructor for instance) and ending in a different routine (C++ destructor). Data region clauses (copy, copyin, copyout) are used to reserve memory on the accelerator device and define data movement between CPU and GPU memories:

- The `copy` clause specifies that data should be copied at region entry from host to device memory and copied back from device to host memory at the end of the region
- The `copyin` clause specifies that data should be copied at region entry from host to device memory, but need not be copied back at the end of the region
- The `copyout` clause specifies that data need not be copied to the device at region entry, but should be copied from device memory to host memory at the end of the region

Note that explicit data directives are not always required even when the host and accelerator use different physical memory spaces. The compiler, for simple access patterns within loops, can often determine which data objects are used and the amount of data to transfer. In these cases, the compiler will either automatically generate data transfers for correct execution of a given compute region or generate a compile-time error if it is unable to do so.

```
void initA(int *A, int N)
{
int i;
#pragma acc parallel loop
    for (i=100; i<N; i++)
        A[i] = -i;
}
```

Example 1: parallel loop without data directives

When compiling the loop above, the PGI OpenACC compiler determines that array 'A' is write-only and that only elements 100 to N-1 of 'A' are written. In such cases the user is not required to specify how to copy data; the compiler will generate an implicit `copyout` directive to copy N-100 elements from the GPU memory location starting at A[100] back to the same elements in the host copy of 'A'. You can see this by compiling the loop in Example 1 using the -Minfo option and targeting NVIDIA Tesla GPUs:

```
% pgcc -ta=tesla -Minfo impex1.c -c
initA:
      4, Accelerator kernel generated
         Generating Tesla code
          5, #pragma acc loop gang, vector(128)
             /* blockIdx.x threadIdx.x */
      4, Generating implicit copyout(A[100:N-100])
```

In some cases the compiler may be unable to determine the amount of data to transfer. Consider this example:

```
int N;

void initA(int *A)
{
int i;
#pragma acc parallel loop
    for (i=100; i<N; i++)
        A[i] = -i;
}
```

Example 2: parallel loop, N global

N is defined as a global variable. The compiler can't guarantee that it won't change during kernel execution, and thus won’t be able to generate an implicit data transfer:

```
% pgcc -ta=tesla -Minfo impex2.c -c
PGC-S-0155-Compiler failed to translate accelerator
region (see -Minfo messages): Could not find
allocated-variable index for symbol (impex2.c: 6)
initA:
      6, Accelerator kernel generated
         Generating Tesla code
          7, #pragma acc loop gang, vector(128)
             /* blockIdx.x threadIdx.x */
      7, Accelerator restriction: size of the GPU
copy of A is unknown
PGC-F-0704-Compilation aborted due to previous
errors. (impex2.c)
```

The scope for generating automatic implicit data transfers is limited to compute regions. Consequently, in many cases using explicit data transfer directives can reduce data movement between host and device memories. In example 3, an implicit copyout of A[100:N-100] will be generated, followed by an implicit copyin of A[0: N]:

```
void initA(int *A, int*B, int N, int M)
{
int i;
/* implicit copy(A[100:N-100]) */
#pragma acc parallel loop
    for (i=100; i<N; i++)
        A[i] = A[i]+i;
/* implicit copyin(A[0:N]) */
#pragma acc parallel loop
    for (i=0; i<N; i++)
        B[i] = A[i];
}
```

Example 3: Redundant OpenACC data movement

Creating a data region that encompasses the two compute regions as shown in Example 4 will eliminate useless data movement between the two compute regions.

```
void initA(int *A, int*B, int N, int M)
{
int i;
#pragma acc copy(A[0:N])
{
#pragma acc parallel loop
    for (i=100; i<N; i++)
        A[i] = A[i]+i;
#pragma acc parallel loop
    for (i=0; i<N; i++)
        B[i] = A[i];
}
}
```

Example 4: Reducing data transfer using data regions

As we’ve shown here, explicit OpenACC data directives are required when the compiler is unable to determine the size of data to transfer, and OpenACC data region directives minimize data transfers between host and device memories.

When porting codes that use deeply nested aggregate data types, adding data directives to specify which members to transfer and keeping their respective live ranges current can be a tedious programming task. In addition, a compute region might access data through multiple indirect references via pointer members of the data structure. Consider the following aggregate data type definitions:

```
struct S1 {
    double* z;
} ;

struct S0 {
    int x;
    struct S1 *B;
} ;
```

As shown in the following code fragment operating on an array A of type S0, a compute region could access *A[i].B->z:

```
#pragma acc parallel loop
    for (i=99; i<899; i++) {
        A[i].x = -i;
        *A[i].B->z = (double)-i;
    }
```

This requires copying A[99:800], then copying each member B of A[i], and then copying each z member of A[i].B. This is known as the ‘deep copy’ problem. [OACC14].

Defining how to copy complicated data structures with succinct directive syntax by adding ‘policies’ to the definition of aggregates is in discussion by the OpenACC committee and is expected in an upcoming version 3.0 of the OpenACC specification. Today, using features in OpenACC 2.6, the programmer must manually copy data by recursively looping through each non-scalar member of an aggregate as shown in the following code:

```
#pragma acc enter data copyin(A[99:800])
    for (i=99; i<899; i++) {
#pragma acc enter data copyin(A[i].B[0:1])
#pragma acc enter data copyin(A[i].B->z[0:1])
    }
```

Updating data on the host requires use of a similar loop structure. It is the responsibility of the programmer to mentally keep track of the live ranges of variables needed in GPU memory. Doing so can be difficult, makes the resulting code harder to maintain, and can be the source of errors whenever a pointer member is assigned and its target is already present or partially present in GPU memory.

2.3 OpenACC and Unified Memory

The OpenACC specification states that "For a shared-memory device, data is accessible to the local thread and to the accelerator. Such data is available to the accelerator for the lifetime of the variable" [OACC17]. When an OpenACC program is compiled targeting a CPU-based SMP system, where there is no need for data copying at compute region boundaries, it is legal for the compiler to ignore any data directives. In fact, because of the statement above in the OpenACC specification, it is legal for the programmer to leave them out entirely. On such a system, OpenACC data management and the difficult task of managing deep aggregate data structures is no longer necessary.

The introduction of CUDA Unified Memory as outlined above in section 2.1 makes this same approach possible on NVIDIA GPU accelerators for allocatable data. PGI OpenACC compilers added this as a production feature in PGI 17.4, with some performance improvements in PGI 17.7 (addition of a host-side pool allocator) and PGI 17.10 (pool allocator interoperability with CUDA Fortran).

When compiling OpenACC for CUDA Unified Memory, the compiler replaces calls to malloc/free in C, operators new/delete in C++ and allocate/deallocate statements in Fortran with cuMemMallocManaged and cuMemFree API calls. Any dynamically allocated data visible to the compiler will be allocated in CUDA Unified Memory and a single address will be used to reference data by the CPU and the GPU.

For program units compiled in this mode, the compiler assumes that any allocatable data object referenced in a Compute Region resides by default in Unified Memory. No error messages will be reported by the compiler in cases where the data size can't be determined. It will generate code assuming the data will be migrated on demand to GPU memory by the CUDA Unified Memory manager.

At execution time, the OpenACC runtime support library performs a dynamic check for each data object referenced in a compute kernel:

- If the data object is in unified memory, the host address is used in the compute region
- If the data object is not in unified memory but is present in device memory, the corresponding device address is used in the compute region
- If the data object is not in unified memory and is not present in device memory, an error is emitted at runtime

A reference in a GPU kernel to a host address that is not in CUDA Unified Memory will lead to an execution error. This situation can happen when a member of an allocatable struct is a pointer that points to a global variable or a stack variable; in the current implementation of CUDA, global, static and stack variables can't be placed in Unified Memory.

3. Using OpenACC and Unified Memory

3.1 The -ta=tesla:managed compiler option

OpenACC for CUDA Unified Memory is enabled by the compile and link-time option -ta=tesla:managed. When this option is used at compile-time, the PGI compilers will intercept and replace all visible or compiler-generated allocates in header files or source files with managed data allocations. When it is used at link-time, it sets the OpenACC runtime libraries to check dynamically whether data is allocated in Unified Memory.

For data in Unified Memory, the runtime routines which implement data movement will essentially do nothing, leaving all data movement to the Unified Memory manager. This has the effect that any data directives in the source code which operate on data objects in Unified Memory are effectively ignored, even though the runtime calls to implement them are still generated by the compiler. This allows mixing of Unified Memory managed data movement and OpenACC programmer directive-based data movement in the same program, which enables use of Unified Memory for all allocatable data in a program while leaving the task of managing global, static or stack data to the programmer.

3.2 Host-side Unified Memory Pool Allocator

Unified Memory managed data allocations allocate host pinned memory as well as device memory and are more expensive than a simple malloc() call. Programs that perform many small data allocations, or which repeatedly allocate and deallocate memory between GPU kernel invocations, can see substantial overhead when these allocations become managed.

One such example is the SPEC ACCEL 1.2 benchmark 356.sp, which in the initial implementation showed a factor of three slowdown when compiled and run with -ta=tesla:managed compared to OpenACC user-directed data management. This slowdown was traced to a Fortran procedure with automatic array dummy arguments, which are dimensioned by an input parameter and must be allocated and deallocated at entry to and exit from the procedure. The procedure included an OpenACC compute region that operated on these arrays, and as a result the newly allocated arrays had to be moved to GPU memory at every invocation of the procedure, resulting in spurious data movement between host and device memory.

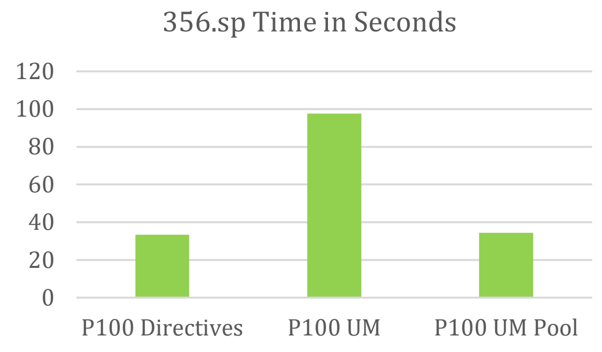


Chart 1: OpenACC Unified Memory Pool Allocator

To address such cases, the PGI compiler enables a host-side Unified Memory pool allocator by default whenever the -ta=tesla:managed option is used. The OpenACC runtime allocates a large pool of managed memory at program startup, and all subsequent allocations are performed out of that pool. In cases like 356.sp, the pool itself tends to migrate to GPU memory and remains there unless the data is referenced on the host. This causes the reallocations to occur from the pool of memory resident on the GPU, significantly reducing overhead. As you can see in Chart 1, performance very close to that obtained with directive-based data movement is regained when the pool allocator is enabled.

The pool allocator is enabled by default with Unified Memory, but it can be disabled or its behavior modified using environment variables. Parameters that can be modified at runtime by programmers include the size of the pool (default is 1GB), the maximum and minimum size of allocations that will be sourced from the pool, and the percentage of total GPU device memory the pool can occupy. [PGI18]

4. Benchmarks Performance

To measure the efficiency of UM-based OpenACC data movement versus user-directed data movement on Piz Daint [CSCS17], we compiled and ran the SPEC ACCEL 1.2 Benchmarks with and without the -ta=tesla:managed option. Each benchmark was compiled with PGI 18.4 and run in the following configurations:

- **P100 Directives** – OpenACC targeting P100 using -ta=tesla:cc60 at compile time
- **P100 UM** – OpenACC targeting P100 in unified mode using -ta=tesla:cc60,managed at compile time, disabling the pool allocator by setting the PGI_ACC_POOL_ALLOC environment variable to zero
- **P100 UM Pool** – OpenACC targeting P100 in unified mode using -ta=tesla:cc60,managed at compile time, pool allocator enabled by default

In Chart 2 the results are normalized to the performance of each benchmark using OpenACC directive-based data movement (P100 Directives) which is always 100%. The P100 UM performance is listed as a percentage of P100 Directives performance.

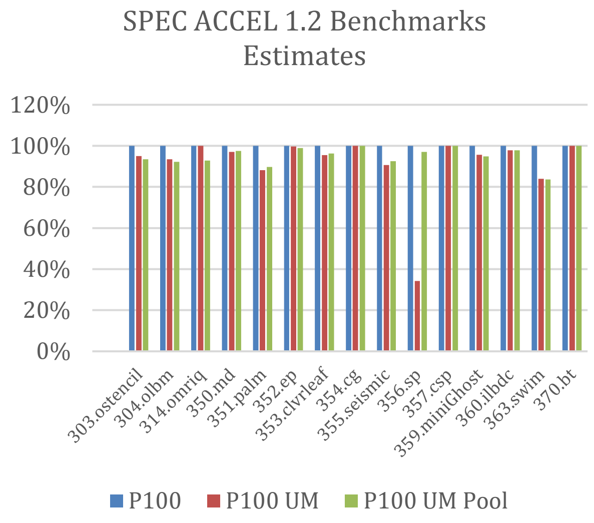


Chart 2: SPEC ACCEL 1.2 Benchmarks on Piz Daint

The results are marked as "estimates" per SPEC rules, which require this designation unless the results are run in the SPEC harness according to SPEC run rules. On average, when the default pool allocator is enabled, the performance of the Unified Memory versions is 95% of performance using explicit directive-based data movement.

To test OpenACC for Unified Memory performance on multiple nodes and GPUs, we compiled and executed the Cloverleaf Mini-App [CUG13] on 1 to 8 MPI ranks on both Piz Daint with P100s and a DGX-1V with Volta V100 GPUs. The results are shown in Chart 3, which displays time in seconds for each run in three different configurations.

- **Base** – Out-of-the box OpenACC version of Cloverleaf as downloaded from the UK-Mac website
- **Base-Managed** – Base version compiled with -ta=tesla:managed to place all allocatable data in CUDA Unified Memory, implicitly enabling use of CUDA Aware MPI and GPUDirect with no code modifications

- **CA** – Base version modified to use OpenACC host_data regions around MPI calls, explicitly enabling use of CUDA Aware MPI and GPUDirect

The initial goal was to compare only the Base and Base-Managed versions, to see how use of Unified Memory versus directives affects scaling on multi-GPU configurations. As shown in Chart 3, while single-GPU performance on DGX-1V was comparable, the multi-GPU performance of the Base-Managed version scaled substantially better than the Base version; the Base version ran in 64.8 seconds on 8 GPUs vs 43.2 seconds for the Base-Managed version.

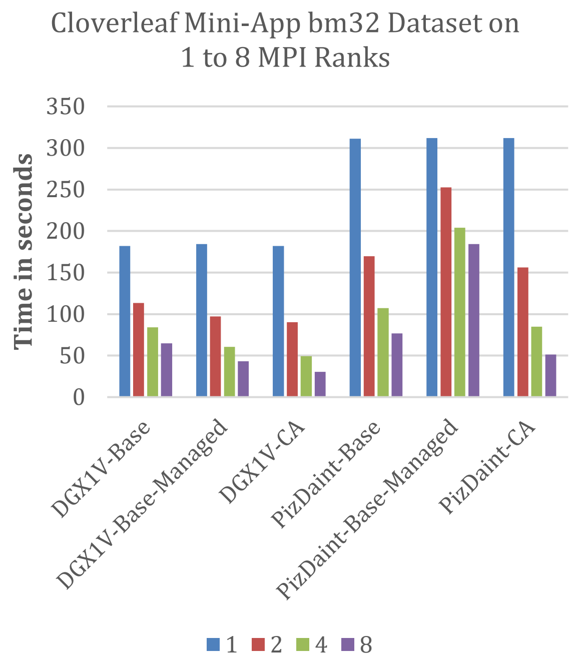


Chart 3: Cloverleaf Mini-App on DGX-1V and Piz Daint

Profiling the two versions showed that MPI calls on allocatable buffers in Unified Memory that occur as part of halo transfers are much more efficient in the Base-Managed version. OpenMPI 1.10.7 interoperates with Unified Memory by disabling CUDA Inter-process Communication (IPC) and GPU Direct RDMA optimizations on Unified Memory buffers [OMPI18], but the pipeline used to implement the MPI calls in this case seems very efficient. The Base version of Cloverleaf with directive-based data movement uses OpenACC update directives before and after the MPI calls to move data from / to accelerator memory to accommodate MPI transfers of host-resident data, which is clearly much less efficient. A simple modification to the OpenACC code using host_data regions to create the "CA" version and enable use of GPUDirect by passing device addresses directly to the MPI calls resulted in even better scaling, reaching a time of 30.6 seconds on 8 MPI ranks accelerated by one V100 GPU per rank.

Running this same sequence of experiments on Piz Daint, we see somewhat slower performance on a single node. This is expected for the P100 GPUs on Piz Daint versus the V100 GPUs on DGX-1V. The scaling of the Base version on 1 to 8 Ranks/Nodes is better than on DGX-1V. The scaling of the CA version improves significantly as well, similar to the result on DGX-1V, indicating that Cray MPI seems to be taking advantage of GPUDirect. The scaling of the Base-Managed version on Piz Daint is poor, indicating that the version of Cray MPI currently installed on Piz Daint is likely not CUDA Unified Memory aware.

The Base-Managed runs outlined above used a version of Cloverleaf with all data directives that apply to allocatables commented out, with a few remaining data directives for non-allocatables. Carrying the experiment one step further, we deleted all OpenACC data directives from the Base version to see if the compiler could implicitly move all non-allocatable data. This version of Cloverleaf, with all data directives removed, successfully compiles and executes with about a 10% performance degradation compared to the Base-Managed version.

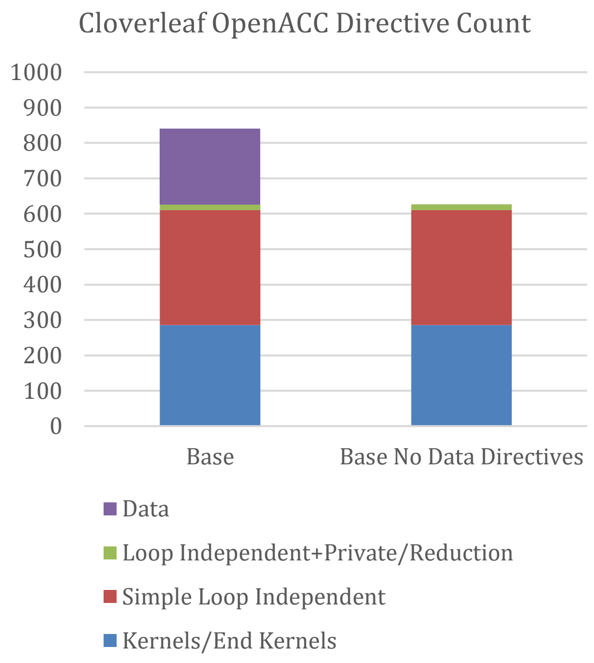


Chart 4: Cloverleaf with and without data directives

About 25% of the overall OpenACC directives in the Base vesion of Cloverleaf are for data management. Eliminating the need for these and the associated work to understand and optimize data movement between host and device memories explicitly can simplify GPU programming and improve developer productivity.

5. Limitations

There are a few limitations to the current PGI implementation of OpenACC and Unified Memory. Perhaps most important is lack of support for static, global or stack data in Unified Memory. This limitation is relatively easy to work around when porting applications that are well-suited to enable mixed use of data directives for these classes of data while relying on Unified Memory for allocatable data. However, complicated allocatable aggregate data objects that include members that point to static data will prevent use of Unified Memory for that object, and in some cases make use of Unified Memory impractical for the entire program. As outlined below, there is a path forward to remove this limitation.

Another concern is the impact on portability of OpenACC programs. If the programmer uses Unified Memory as an easy on-ramp and porting tool, followed by insertion of explicit data directives, the resulting code will be portable to any OpenACC compiler. However, if no data directives are added to the code and it relies on shared memory between host and accelerator, that may limit portability to other compilers and systems.

The current implementation in effect completely ignores data directives that are present in the code. An obvious improvement would be to interpret any data directives as hints on when to move data or where to place it by default.

Finally, we have only done limited testing of the Unified Memory oversubscription feature, which allows for data sets that are larger than the physical GPU memory. While OpenACC and Unified Memory oversubscription should work as expected, we need to do more testing of this capability for both robustness and performance.

6. Conclusions and Future Directions

OpenACC and CUDA Unified Memory can work together to simplify the task of accelerating applications using NVIDIA Tesla GPUs, allowing programmers to focus on exposing parallelism rather than on details of data management. The performance of OpenACC programs relying partially or exclusively on Unified Memory for data movement between host and device memories is usually nearly as good and sometimes better than user-directed data movement. The inherently higher overhead of Unified Memory data allocation can be mitigated in many cases using a host-side pool allocator.

We expect future releases of the PGI compilers and CUDA to support all classes of data in Unified Memory on x86-64 CPU-based systems by leveraging the Linux Heterogeneous Memory Manager (HMM) [GTC17]. We also expect future releases of the PGI compilers compiling in Unified Memory mode will map data directives to the cuMemAdvise and cuMemPrefetch APIs, giving the programmer the ability to rely primarily on Unified Memory but fine-tune data movement as needed.

About the Authors

Sebastien Deldon is a senior compiler engineer in the PGI compilers & tools group at NVIDIA; since he joined PGI in 2004, he has been working on several aspects of PGI compilers and tools, and is currently focused on GPU code generation (OpenACC/CUDA Fortran) since 2013; He can be reached by e-mail at sdeldon@nvidia.com.

James Beyer is a senior engineer in the CUDA software group at NVIDIA; as a 15-year veteran of the Cray Compilation Environment optimization team he has been involved in the design and implementation of OpenACC from the beginning. He can be reached by e-mail at jbeyer@nvidia.com.

Doug Miles is director of PGI compilers & tools at NVIDIA. He can be reached by e-mail at dmiles@nvidia.com.

References

[CSCS17] Centro Svizzero di Calcolo Scientifico (CSCS), *"Piz Daint", one of the most powerful supercomputers in the world*, www.cscs.ch, March, 2016.

[CUG13] A.C. Mallinson, D.A. Beckingsale, W.P. Gaudin, J.A. Herdman, J.M. Levesque, S.A. Jarvis, CloverLeaf: Preparing Hydrodynamics Codes for Exascale, cug.org, The Cray User Group 2013, May 6-9 2013.

[GTC16] Nikolai Sakharnykh, *The Future of Unified Memory*, presentation at GTC 2016 on-demand.gputechconf.com, April 2016.

[GTC17] John Hubbard, *Using HMM to Blur the Lines between CPU and GPU Programming*, on-demand.gputechconf.com, May 2017.

[GTC18] Stefan Maintz, Markus Wetzstein, *Porting VASP to GPUs with OpenACC*, on-demand.gputechconf.com, March 2018.

[HPCW17] John Russell, *OpenACC Shines in Global Climate/Weather Codes*, www.hpcwire.com, November 2017.

[NV16] NVIDIA Corp., *NVIDIA Tesla P100*, 2016.

[NV18] NVIDIA Corp., *CUDA C Programming Guide*, docs.nvidia.com, May 2018.

[OACC14] OpenACC Architecture Review Board, *Complex Data Management in OpenACC Programs Technical Report TR-14-1*, www.openacc.org, November 2014.

[OACC15] OpenACC Architecture Review Board, *OpenACC Programming and Best Practices Guide*, www.openacc.org, June 2015.

[OACC17] OpenACC Architecture Review Board, *OpenACC Application Programming Interface Version 2.6* www.openacc.org, November 2017.

[OMPI18] Open MPI FAQ, *Running CUDA-aware Open MPI*, www.open-mpi.org, January 2018.

[PGI18] The Portland Group, *PGI Compilers and Tools User's Guide*, www.pgroup.com April 2018.

[UM13] Mark Harris, *Unified Memory in CUDA 6*, devblogs.nvidia.com, November 2013.