
OpenACC
PGI®

OpenACC for Multicore CPUs

by Michael Wolfe, PGI Compiler Engineer

OpenACC is designed as a parallel programming model that can deliver high performance on a wide range of systems, including accelerated systems with GPUs, multicore CPUs, and manycore processors. Until recently, PGI has focused its OpenACC development efforts on the NVIDIA Tesla and AMD Radeon GPU targets. Performance on these two different GPUs is comparable using OpenACC, but until now performance portability of OpenACC on conventional multicore microprocessors has not been demonstrated. That changes with the latest release of the PGI Accelerator compilers.

The latest PGI Accelerator compilers allow you to generate parallel code from your OpenACC programs for Intel and AMD multicore processors. This feature is enabled with a new option to the **`-ta`** (target accelerator) command line flag, **`-ta=multicore`**. Adding this option to the command line tells the compiler to generate parallel multicore code for OpenACC compute regions, instead of the default of generating parallel GPU kernels. The parallel multicore code will execute in much the same fashion as if you had used OpenMP **`omp parallel`** directives instead of OpenACC compute regions.

This feature raises a number of questions:

- How do I use OpenACC for multicore CPUs?
- Why should I use OpenACC for multicore CPUs?
- Is data still copied for the OpenACC data clauses?
- What is the performance of OpenACC relative to OpenMP?
- Can I now use OpenACC to spread work across a GPU and the CPU cores?
- Can I create a single binary that runs on either a GPU or a multicore CPU?
- How does OpenACC for multicore CPUs interoperate with OpenMP?
- Does asynchronous execution work with OpenACC for multicore CPUs?
- Does OpenACC for multicore CPUs make OpenMP less important?
- Can OpenACC for multicore CPUs be used on Intel Xeon Phi?
- Do I need a GPU-enabled PGI license to use OpenACC on multicore CPUs?
- How does OpenACC for multicore CPUs relate to the –ta=host option?
- Does OpenACC for multicore CPUs work on Windows and OS X?
- Are there limitations, and what improvements can I expect in the future?
- When will OpenACC and OpenMP merge?

The rest of this article will address these questions.

How do I use OpenACC for multicore CPUs?

It's as simple as it looks. Add the **-ta=multicore** flag to your pgfortran, pgc++ or pgcc command line, both for the compile and link steps. You can also specify **-acc**, which enables the OpenACC directives, but the **-ta** flag implies **-acc**, so it's not strictly necessary. Adding **-Minfo** or **-Minfo=accel** will enable PGI compiler feedback messages, giving details about the parallel code generated, such as:

```
ninvr:
      59, Loop is parallelizable
          Generating Multicore code
      59, #pragma acc loop gang
pinvr:
      90, Loop is parallelizable
          Generating Multicore code
      90, #pragma acc loop gang
```

You can control how many threads the program will use to run the parallel compute regions with the environment variable **ACC_NUM_CORES**. The default is to count how many cores are available on the system. For Linux targets, the runtime will only count physical cores (not hyper-threaded logical cores), and use that many threads. The OpenACC gang-parallel loops will be run in parallel across the threads. If you have an OpenACC parallel construct with a num_gangs(200) clause, the runtime will take the minimum of the num_gangs argument and the number of cores on the system, and launch that many threads. That avoids the problem of launching hundreds or thousands of gangs, which makes sense on a GPU but which would overload a multicore CPU.

Why should I use OpenACC for multicore CPUs?

This gets to the basic reason why OpenACC was created four years ago. Our goal is to have a single programming model that will allow you to write a single program that runs with high performance in parallel across a wide range of target systems. Until now we have been developing and delivering OpenACC targeting NVIDIA Tesla and AMD Radeon GPUs, but the performance-portability story depends on being able to demonstrate the same program running with high performance in parallel on non-GPU targets, and in particular on a multicore host CPU. The first reason to use OpenACC with **-ta=multicore** is if you have an application that you want to use on systems with GPUs, and on other systems without GPUs but with multicore CPUs. This allows you to develop your program once, without having to include compile-time conditionals (ifdefs) or special modules for each target with the increased development and maintenance cost.

Even if you are only interested in GPU-accelerated targets, you can do parallel OpenACC code development and testing on your multicore laptop or workstation without a GPU. This can separate algorithm development from GPU performance tuning. Debugging is often easier on the host than with a heterogeneous binary with both host and device.

Is data still copied for the OpenACC data clauses?

Short answer: no.

The longer answer is that in the OpenACC execution model, the multicore CPU is treated like an accelerator device that shares memory with the initial host thread. With a shared-memory device, most of the OpenACC data clauses (copy, copyin, copyout, create) are ignored, and the accelerator device (the parallel multicore) uses the same data as the initial host thread. Similarly, update directives and most OpenACC data API routines will not generate data allocation or movement. Other data clauses are still honored, such as private and reduction, which may require some dynamic memory allocation and data movement, but no more than the corresponding OpenMP data clauses.

When using OpenACC with a GPU, data is copied from the system memory to device memory (and back). The user is responsible for keeping the two copies of data coherent, as needed. When using OpenACC on a multicore CPU, there is only one copy of the data, so there is no coherence problem. However, the GPU-OpenACC program can produce different results than a multicore-OpenACC program, if the program depends on the parallel compute regions updating a different copy of the data than the sequential initial host thread regions.

```
#pragma acc data create(a[0:n]) present(x[0:n],b[0:n])
{
// following loop executed on device
    #pragma acc parallel loop
    for(i=0;i<n;++i) a[i] = b[i];

// following loop executed on host
    for(i=0;i<n;++i) a[i] = c[i];

// following loop executed on device
    #pragma acc parallel loop
    for(i=0;i<n;++i) x[i] = a[i];

    ...
}
```

On a GPU, the above code fragment allocates a copy of the array **a** on the device. It then fills in the device copy and the host copy with different values. The last loop will get the values from the device copy of **a**, so it's equivalent to **x[i]=b[i]**. When compiled for a multicore, the first two loops are both executed on the CPU, the first with a single thread and the second with all multicore threads. Both loops update the same copy of **a**, and the last loop will be equivalent to **x[i]=c[i]**.

What is the performance of OpenACC relative to OpenMP?

The PGI OpenACC multicore CPU runtime uses essentially the same code generation and thread management that our compilers have used for OpenMP and autoparallelization for the past 20 years. Our experiments show that the OpenACC multicore performance is equivalent to the same loop(s) running in parallel with OpenMP.

The more pertinent question is whether an OpenACC version of a program will reach the same (or better or worse) performance than an OpenMP version of a program. A typical large OpenMP application will have a single outer parallel construct in an outer routine, then a number of explicit or implicit barrier operations within the program. A typical large OpenACC application will have a larger number of parallel constructs, partly because OpenACC has no barrier across all parallel gangs. For some programs, the OpenMP multicore program can have better performance because there is less thread creation and coordination. For other programs, the OpenACC multicore program can have better performance, because there is less redundant execution and even less synchronization.

Can I now use OpenACC to spread work across a GPU and the CPU cores?

Short answer: no.

The longer answer is this feature treats the multicore like an accelerator device. Spreading the work across a GPU and the multicore is like spreading the work across multiple GPUs, that is, across multiple devices. There are many issues with parallel execution on a single device, but across multiple devices, especially heterogeneous devices, is a particularly difficult problem. With current and foreseeable devices, the problems are as much about data as about compute: data distribution and replication across device memories, data coherence, work distribution, load balancing, and more. The OpenACC technical committee is looking at how to make multiple device support more natural, but that will take some significant work and creativity.

Can I create a single binary that runs on either a GPU or a multicore CPU?

The current release does not support a unified **`-ta=tesla,multicore`** or **`-ta=radeon,multicore`** binary. We plan to support this feature in a future release, allowing you to build a single binary that will run in parallel on a GPU when one is present, and run in parallel on the multicore otherwise. This could be used on a cluster where some of the nodes are GPU-accelerated and other nodes are not, for instance. Alternatively, you can build a single binary to deliver to users or customers where some have GPU-accelerated systems and others do not.

How does OpenACC for multicore CPUs interoperate with OpenMP?

The PGI compilers support OpenACC compute constructs inside OpenMP parallel regions when targeting the OpenACC code for a GPU, whether the parallel OpenMP threads are sharing a GPU or each using a different GPU. However, the parallel code generated for **`-ta=multicore`** is very similar to the parallel code generated for OpenMP parallel loops. If you are using OpenMP to distribute work across the CPU cores, there is little need for OpenACC to also spread work across the same cores. In fact, generating too many threads will oversubscribe the cores and could even slow the program down. This release does not support OpenACC compute constructs targeted at a multicore within OpenMP parallel regions.

Does OpenACC for multicore CPUs make OpenMP less important?

Certainly not. OpenMP has been widely and successfully used for multiprocessor and multicore systems and nodes for shared memory parallelism, most typically with relatively small processor and core counts. It includes loop and functional parallelism, and has a wide range of synchronization constructs. It is supported on essentially all current multicore systems in the HPC space. The directives are explicitly designed such that a mechanical process can translate the directives and constructs into parallel code, with explicitly defined behavior.

OpenACC is designed for highly parallel computations. It focuses on loop parallelism and has few scalability-limiting synchronization constructs. The directives are designed such that the code can be efficiently compiled for a wide range of parallel systems with very different parallelism profiles. The two languages have different goals, and we expect they will be used together.

Does asynchronous execution work with OpenACC for multicore CPUs?

When targeting a GPU, the compute regions and data movement can be done asynchronously from host computation, allowing yet another level of parallelism. When targeting the multicore itself, there is no data movement, and the compute regions are executed on the same multicore as the host computation, so asynchronous execution doesn't make much sense.

Can OpenACC for multicore CPUs be used on Intel Xeon Phi?

The PGI Accelerator compilers do not support Xeon Phi. We are planning to optimize the PGI compilers for manycore x86 processors when Knights Landing systems become available, including tuning our OpenACC and OpenMP implementations for those processors. One of the more interesting features of the Knights Landing is the exposed memory hierarchy, the near (high bandwidth) and far (system) memories. We will be studying how to use the OpenACC data constructs to manage data movement between those two memories.

Do I need a GPU-enabled PGI license to use OpenACC on multicore CPUs?

This feature will work with any valid PGI license that includes access to the PGI 15.10 or later versions of compilers. It will work on any supported PGI multicore CPU target platform – Linux, Windows or OS X – including those for which no GPU target is supported.

How does OpenACC for multicore CPUs relate to the –ta=host option?

The PGI OpenACC compilers have always supported the **`-ta=host`** option, along with **`-ta=tesla,host`** (or **`-ta=radeon,host`**). The former option would generate single-threaded code for the OpenACC regions, essentially ignoring the directives. The latter options would generate code which would use a GPU when one is available, and would execute sequential code on the host when there is no GPU. These are still supported in this release in the same way as before. In the future, we may keep this feature, or roll it into the multicore target, depending on experience with OpenACC nested inside OpenMP parallel regions.

Are there limitations, and what improvements can I expect in the future?

There are a few limitations in the initial release, which we expect to remove in future releases. In initial releases of the PGI compilers supporting this feature the collapse clause is ignored, this means only the outer loop is parallelized. The worker level of parallelism is ignored; we are still investigating how best to generate parallel code that includes gang, worker and vector parallelism. Also, no loop code optimization or tuning is done. For example, when compiling for a GPU, the compiler will reorder loops to optimize array strides for the parallelism profile. None of these features have been implemented for the multicore target in this release. Additionally, the vector level of parallelism will be used to generate SIMD code in a future release but we are not doing so now. Application performance should improve as these limitations are addressed.

As mentioned above, we plan to do further work on optimizing and tuning OpenACC multicore code generation and to study the interaction with OpenMP. As the core counts increase, you might use OpenMP for coarse-grain parallelism, either across sockets or across partitions of the cores. Then you might use nested OpenACC compute constructs for fine-grain parallelism, within a socket or a partition. We will also be looking at new targets including IBM OpenPOWER CPUs and Intel Knights Landing manycore processors.

When will OpenACC and OpenMP merge?

The upcoming release of the OpenMP specification adopts many features from OpenACC, such as unstructured data lifetimes (enter data, exit data), asynchronous device computation (though OpenMP does this through task parallelism instead of async queues), and API routines for device data allocation and management. The implementation of multicore code for OpenACC seems to bring the two closer together. So, are there any fundamental differences between OpenMP and OpenACC that prevent an immediate merger?

There are two types of constructs in OpenACC and OpenMP for heterogeneous computation: data and compute. The data constructs manage data allocation in device memory, movement between system and device memory, and the correspondence of data between the two. While the spellings are different, and not all features in either language appear in the other, the two data models are essentially coherent. This means a single implementation could support both the OpenACC and the OpenMP data models, and OpenACC data directives could be ported to OpenMP, and vice versa, with little effort.

The compute constructs are somewhat more different. Both OpenACC and OpenMP support three levels of parallelism, gang, worker, vector in OpenACC, and team, thread, simd in OpenMP. OpenACC was designed with GPUs, multicore, manycore, and other targets in mind, meaning the mapping of language parallelism to hardware parallelism is not strict. The PGI OpenACC implementation will map gang parallelism across threads on a multicore, and across thread blocks on an NVIDIA GPU. OpenMP distinguishes between host multicore parallelism (only thread and simd) and target device parallelism (team, thread, and simd). OpenMP was originally designed with only thread parallelism, and there is little experience with team (or simd) parallelism. If you write your OpenMP program with only thread parallelism, which has many implicit barriers in addition to any synchronization you include, your program will not be able to take advantage of all the parallelism on a GPU. If you write your OpenMP program with team parallelism, it's not clear how any implementations will support that on a multicore system, particularly since team parallelism is only supported within a target region.

We feel that parallelism within a socket and a node will increase rapidly over the coming years, so we need a fundamentally scalable, modern programming model. Different vendors will have different solution points, meaning support for different types of parallelism and tradeoffs between them. We think the OpenACC compute model is a step in the right direction for now and for the future. If and when OpenMP adopts such a compute model, the time may be right for the languages to merge.

Summary

The PGI 15.10 release allows you to generate multicore CPU code from your OpenACC parallel programs. This can be used to write truly performance-portable parallel programs, without spending a lot of time recoding to target both GPU-accelerated and multicore systems. Use the **`-ta=multicore`** option to enable this feature, and we always recommend enabling **`-Minfo=accel`** for the compiler feedback messages.

OpenACC and CUDA Unified Memory

by Michael Wolfe, PGI Compiler Engineer

One of the most important issues when programming a system with a GPU or any attached accelerator is managing data movement between host memory and device memory. This is a special case of managing a memory hierarchy, a problem that has been with us since the dawn of computing. The very first computer I programmed was an IBM 360/75 at the University of Illinois which had a whole megabyte of magnetic core memory, composed of 400KB of fast core and 600KB of slow core. Systems in those days didn't support virtual memory, so programmers were responsible for staging data from disk (or tape) storage to memory and back, and for telling the operating system the maximum amount of memory the program would use (or on this system, the maximum amount of fast and of slow core that the program would use).

Introduction

Since that time, much of the memory hierarchy management problem has been relegated to the hardware and operating system. Common CPUs now include fast on-chip cache memories with hardware to detect cache misses, load the missed data into the cache, evict stale data, and detect conflicts between caches attached to different cores or processors. Programmers now mostly ignore cache memory, but sometimes will tune a program to enhance locality, thus improving the cache hit rate and the total performance. Operating systems use address translation hardware to support paged virtual memory, detecting when a program needs more memory than is physically available, evicting stale pages to disk and reloading pages when needed. Some decades ago, programmers would sometimes optimize for virtual memory locality in the same way that they optimize today for caches. Now that current systems have many gigabytes of physical memory, the need for virtual memory locality is much reduced except for those applications with extremely large datasets.

However, today's attached accelerators, such as GPUs, bring us back to a programmer-managed memory hierarchy. Using CUDA or OpenCL, the programmer is responsible for allocating memory either in the host memory or in the device memory, and for copying data between the two. This is an extra burden placed on the programmer for two reasons. First, the memory requirements and technologies for host memory and device memory are quite different. The host already has a very large memory, quite often 16GB, 32GB, 64GB or larger. This memory doesn't need to be particularly high performance, since the CPU has a large, deep cache hierarchy. With such a large memory, cost is key. A highly parallel device like a GPU requires a much higher bandwidth memory system. Current GPUs use GDDR, which supports memory bandwidths of over 300GB/s in the right configuration, which is several times faster than CPU memory bandwidth. CPU performance depends on most memory accesses hitting in the cache memory, so a lower main memory bandwidth is sufficient because it only needs to satisfy the cache misses. Highly parallel devices like GPUs are designed to work efficiently with very large datasets where cache memories are

too small to be effective, so they require higher memory bandwidths. This memory is more expensive and therefore smaller. The smaller size often requires data to be shuffled from the host memory to device memory and back again, much like data was moved between CPU and disk some four decades ago.

The second reason this problem is left for the programmer is that automating the memory hierarchy management requires more hardware support than is currently available on these devices. We might think of the device memory like a cache on the device, backed by the system memory. What would it take to manage this just like today's hardware caches, or like today's virtual memory? Unfortunately, this requires several hardware innovations that exist in today's CPUs, but are not available in current GPUs, such as page translation, miss detection, kernel suspend and resume, and more.

However, NVIDIA GPUs and CUDA drivers support a feature called CUDA Unified Memory, which allows a program to dynamically allocate data that can be accessed from the CPU or from the GPU with the same pointer and address. The CUDA driver will ensure that the data is in the right memory at the right time. The PGI CUDA Fortran compiler added support for a **`managed`** memory attribute last year. Now we have implemented a feature to allow you to use CUDA Unified Memory seamlessly in OpenACC programs. This article describes how to use the feature and what behavior to expect, and some problems and pitfalls you might encounter on the way. Internally, we have found this feature to be a great help when initially porting a program to NVIDIA GPUs with OpenACC, by allowing us to focus first on porting the compute regions and then tuning the data movement afterwards.

Using CUDA Unified Memory

The command line option to enable CUDA Unified Memory with the PGI OpenACC compiler is **`-ta=tesla:managed`**. This feature is only supported on 64-bit Linux targets. This should be specified when compiling and linking your program. This option changes the way dynamic memory is allocated. In particular, for C programs, it changes the **`stdlib.h`** header file so that calls to **`malloc`**, **`calloc`** and **`free`** are changed (using #**`define`**) to new routines that allocate using CUDA Unified Memory. Your C source program must include **`stdlib.h`** for this to work. For C++ programs, it changes the **`cstdlib`** header file in the same way. Also, for C++ programs, it changes **`new`** and **`delete`** to allocate using CUDA Unified Memory. For Fortran programs, **`allocatable`** arrays are changed to include the CUDA Fortran **`managed`** attribute, so that allocating these arrays will use CUDA Unified Memory.

The **`-ta=tesla:managed`** option must also be used at link time. This sets the OpenACC runtime to test at execution time whether data was allocated in CUDA Unified Memory. If so, then the OpenACC runtime does nothing, letting the CUDA driver manage data movement between the system and device memories. Otherwise, the OpenACC runtime proceeds normally. This allows mixing driver-managed CUDA Unified Memory and OpenACC runtime-managed data in the same program which can be very handy.

Expected Behavior

Let's take a very simple test program and see the difference in behavior.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]){
    float *a, *b;
    int n, i;

    n = atoi( argv[1] );

/* allocate */
    a = (float*)malloc( sizeof(float)*n );
    b = (float*)malloc( sizeof(float)*n );

/* initialize */
    for( i = 0; i < n; ++i ){ a[i] = i; b[i] = 2*i; }

/* run a parallel loop */
    #pragma acc parallel loop
    for( i = 0; i < n; ++i ) a[i] *= b[i];

/* print partial results */
    printf( "%f %f %f\n", a[0], a[1], a[2] );
    printf( "%f %f %f\n", a[n-3], a[n-2], a[n-1] );

    return 0;
}
```

First, compile this with OpenACC, using **pgcc -acc**. Run the binary after setting the environment variable **PGI_ACC_NOTIFY** to 3; this will print out a runtime diagnostic after each data movement and each GPU kernel launch. If you run this program with an argument of 1000, you will see output like:

```
upload CUDA data file=bb.c function=main line=17 device=0
       variable=a bytes=4000
upload CUDA data file=bb.c function=main line=17 device=0
       variable=b bytes=4000
launch CUDA kernel file=bb.c function=main line=17 device=0
       num_gangs=4 num_workers=1 vector_length=256 grid=4
       block=256
download CUDA data file=bb.c function=main line=19 device=0
         variable=a bytes=4000

0.000000 3.000000 6.000000
2991.000000 2994.000000 2997.000000
```

This shows that the compiler implicitly added a **copy** clause for the array **a** and a **copyin** clause for **b**. If you recompile enabling this new feature, using **pgcc -acc -ta=tesla:managed**, and then run the program, the output will include the launch line and the output, but no data movement:

```
launch CUDA kernel file=bb.c function=main line=17 device=0
       num_gangs=4 num_workers=1 vector_length=256 grid=4
       block=256

0.000000 3.000000 6.000000
2991.000000 2994.000000 2997.000000
```

You will see the same behavior even if you add explicit data clauses for the arrays **a** and **b**. Because the arrays get allocated using CUDA Unified Memory, the OpenACC runtime lets the CUDA driver manage data movement. Because the data movement is not managed by the OpenACC runtime, it is not reported.

The Details

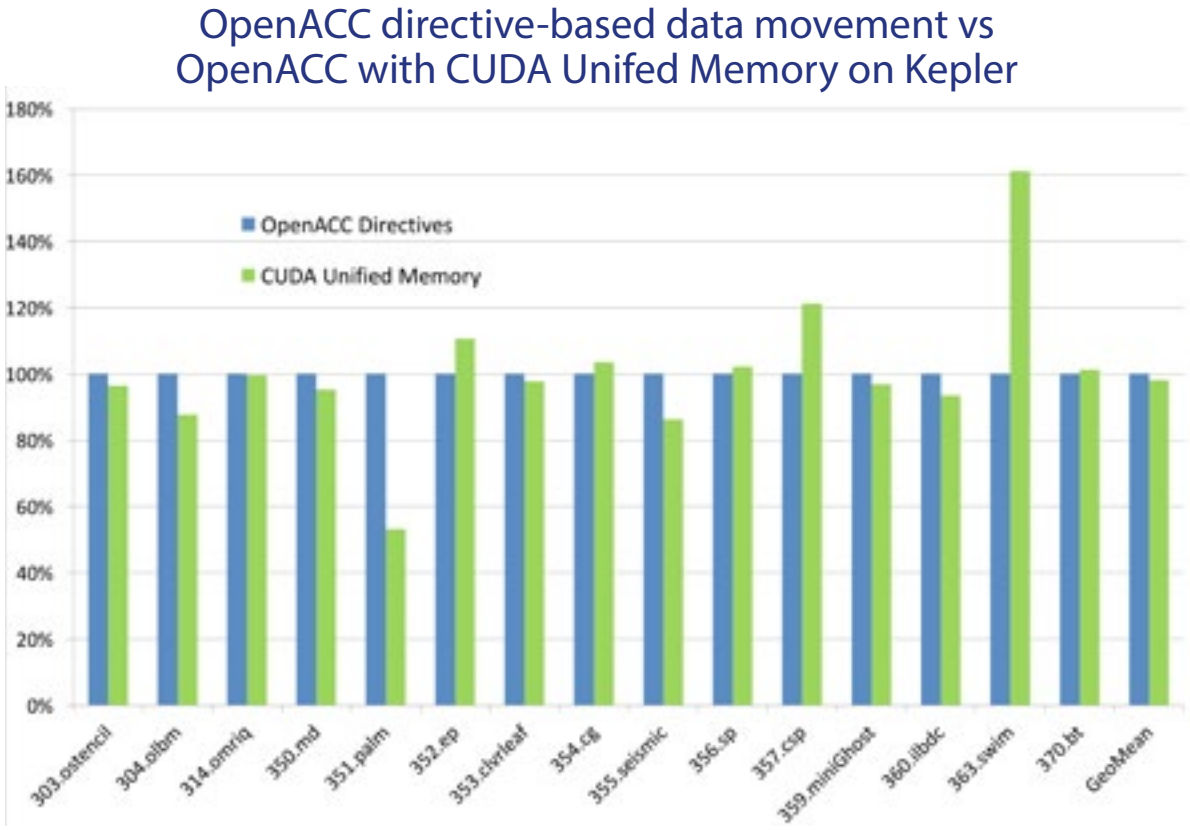
It looks simple, just adding a command line option, and in many cases it really is that simple. However, it's not perfect, and it's not ready for production use in many cases. There are some details of the implementation that you should be aware of and some problems that you may run into. Some of these problems will be addressed in future PGI releases as we improve this feature, but some will have to wait for future NVIDIA GPU architectures, as described earlier.

- We support this feature only on NVIDIA Kepler and later GPUs.
- Data is moved to the device only at kernel launches. This means you can't hide data movement behind other host computation, and you can't prefetch data to the device memory.
- Data motion is synchronous. This is somewhat misleading, since the actual data motion is done using fast pinned asynchronous data transfers, but from the perspective of the program the transfers are synchronous. The transfers aren't overlapped with any user code.
- Each OpenACC parallel region or kernels region generates one or more GPU kernels. When a GPU kernel is launched, all the CUDA Unified Memory is moved to the device. If the data is already present on the device, it doesn't need to be moved again, but even data that isn't used in a kernel or an OpenACC compute region is moved to the device. This is necessary because there is no hardware on current GPUs to detect the equivalent of a cache miss or page fault.
- It only works for dynamically-allocated memory. Static data (C static and extern variables or Fortran module, common block and save variables) and function local data will still be managed using the OpenACC runtime.
- The **`-ta=tesla:managed`** flag must be specified for the file which allocates the variable, even if there is no OpenACC code in that file. The CUDA Unified Memory manager can only manage data that is allocated using the **`cudaMallocManaged`** routine.
- When the program allocates managed memory, it actually allocates host pinned memory as well as device memory. This means the total amount of managed memory is limited to the available device memory.
- Because the host allocation uses pinned memory, the allocate and free operations are somewhat more expensive. It also means the data transfers are somewhat faster. This means comparing performance for using managed memory vs. the normal OpenACC data management can be misleading. We have seen examples where using managed memory gives a performance improvement, because data transfers to host pinned memory are so much faster.
- C++ virtual function tables don't work.
- Kernel launches are actually asynchronous. If the host program continues and accesses managed data while a GPU kernel is still executing, the program can fail with a segfault. This can occur even if the host and device are accessing completely different data. Remember that all managed data is moved to the device memory at a kernel launch.


- It only works with a single GPU. The current release only supports device zero, but we are working on lifting that restriction.
- Not all the allocation routines are available. There is no C version of **`realloc`**, for instance.
- In Fortran, it only currently works for allocatable arrays (not array pointers).
- It is only supported with CUDA 6.5 and later releases and device drivers, and only on 64-bit Linux.

Performance

One of the big challenges using accelerator devices today is data management. It would really simplify life for a programmer if the system hardware and system software could automatically manage the device memory as if it were a cache, or similar to virtual memory. Such support will come in future GPUs, but that raises the question today whether the performance of automatic data management will be satisfactory, or whether user data management will always be required. Using CUDA Unified Memory today gives a first hint at an answer, and it's quite promising.



As an example, we compiled and executed the 15 OpenACC SPEC ACCEL V1.0 benchmarks with and without the `-ta=tesla:managed` option. The numbers shown in the chart reflect relative performance of using the managed memory feature versus the default OpenACC execution mode in which OpenACC directives specify when and where data is moved back and forth between host and GPU memory. These were compliant SPEC ACCEL runs on an Intel Sandy Bridge CPU with an NVIDIA K40 GPU.

The performance overall is quite good. A few of the benchmarks show a speedup using managed memory. This is a side effect caused by the fact that managed memory uses pinned host memory for data transfers, whereas the default for the PGI OpenACC compiler is not to place any data in pinned memory. The 351.palm benchmark exhibits a pretty serious slowdown. This is because an FFT operation must be performed on the host after every time step on the GPU, resulting in data movement back and forth at every time step. This data movement is optimized in the default OpenACC version using appropriate data directives.

In most of these benchmarks, all data movement is managed using CUDA Unified Memory. These results indicate that locality works; when most of the compute operations are being done on the GPU, the data can stay resident on the GPU and data movement will not be the limiting factor in performance. For the cases where we see performance degradations, future hardware support will help because not all the managed data will have to be moved before each kernel launch. However, we expect that there will always be cases where a user will want to control data movement, for instance to prefetch data to the right memory and overlap that with other computation.

Summary

PGI's OpenACC and CUDA Unified Memory feature allows you to experiment with OpenACC and managed memory. It can greatly simplify initial porting of applications to OpenACC, and data management overall, modulo the limitations outlined above. We have many users who now use this feature regularly for initial porting, and performance results so far indicate that the combination of OpenACC and CUDA Unified Memory on future GPUs will enable a much simpler interface for programming accelerated computing platforms.

Fortran Array Attributes: Pointer and Allocatable, Contiguous and Target

by Michael Wolfe, PGI Compiler Engineer

Fortran has some nice high level features that allow arrays to be treated as objects, including multidimensional arrays, array assignments, arrays and array sections as arguments, and more. Dynamically allocated arrays can be created with the pointer or allocatable attributes:

```
real, dimension(:), allocatable :: a
real, dimension(:), pointer :: p
...
allocate( a(n), p(n) )
```

It may seem like allocatable arrays and pointer arrays provide the same functionality, and compilers are likely to implement pointer and allocatable arrays with the same pointer and descriptor. However, there are three significant differences between these that affect performance, and you should take these into account when developing Fortran programs.

Aliasing

The first difference is that an allocatable array can only be created with an allocate statement. Two allocatable arrays can never alias each other:

```
real, dimension(:), allocatable :: a, b
allocate( a(n), b(n) )
...
a(:) = b(:) + 2
```

The compiler knows that the array **a** is distinct from the array **b**, so the array assignment can be simply flattened to a loop. An array pointer can be filled in with an allocate statement, or with a pointer assignment. Two array pointers can alias each other, sometimes in strange ways:

```
real, dimension(:), pointer :: a, b, c
allocate( c(n+1) )
a => c(2:n+1)
b => c(1:n)
...
a(:) = b(:) + 2
```

In this example, `a(1)` aliases `c(2)`, as does `b(2)`. Flattening the array assignment to the loop:

```
do i = 1, n
   a(i) = b(i) + 2
enddo
```

This recurrence relation is not a legal implementation of the array assignment, which requires evaluation of the right hand side for all indices before any assignments. Implementing this correctly is usually done by creating a temporary array to hold the right hand side values:

```
allocate( temp(1:n) )
do i = 1, n
   temp(i) = c(i) + 2
enddo
do i = 1, n
   c(i+1) = temp(i)
enddo
deallocate( temp )
```

This adds overhead to manage the temp array, and adds an extra copy loop. In this example, an aggressive enough compiler could substitute the pointer assignments, but in general the array pointers are assigned in some other routine in some other file, so the compiler must assume the worst case.

While array pointers can alias other array pointers, an array pointer cannot alias another array, including another allocatable array, unless that array has the target attribute. The compiler can do a better job of optimization when it can refine the array aliases more precisely, so minimizing the use of array pointers when they are really allocatables helps significantly.

Contiguity

When an array is created, whether it be a static module array, a fixed size or deferred-shape local array, or a dynamically allocated array, the array is contiguous. All the elements of the array are in a single block of memory. For a one-dimensional array, `a(1)` will be next to `a(2)`, and so on. Since allocatable arrays can only be created with an allocate statement, an allocatable array is always contiguous.

Array pointers are not always contiguous:

```
real, dimension(:), allocatable :: a
real, dimension(:), pointer :: b, p1, p2
real, dimension(:,:), pointer :: c
...
allocate( a(n),b(n),c(n,n) )
p1 => b(1:n:2)
do i = 1, n
   p2 => c(i,:)
   ...
   call foo( a, p1, p2 )
enddo
```

At the call statement, the compiler knows that **a** is a contiguous array, because it is allocatable. However, the pointer arrays **p1** and **p2** are not. The elements **p1(i)** and **p1(i+1)** are two apart; we say the stride is two. The elements **p2(i)** and **p2(i+1)** are **n** elements apart, so the stride is **n**.

Whether an array is contiguous or not can be important at subroutine or function calls. If the subroutine interface uses FORTRAN77-style array dummy arguments, either fixed size, assumed-size, or deferred-shape, the actual argument must be contiguous. If the actual argument is not or might not be contiguous, the compiler is required to copy the actual argument array to a contiguous temporary array, or to insert a runtime test for whether the actual argument is contiguous and copy the array if not:

```
subroutine sub( x )
real :: x(*) ! assumed-size dummy argument
             ! actual argument must be contiguous
...
end subroutine
...
real, dimension(100) :: a
real, dimension(:), allocatable :: b
real, dimension(:), pointer :: p
...
call sub( a )         ! a is contiguous
call sub( b )         ! b is allocatable, is contiguous
call sub( b(1:n:2) ) ! section is not contiguous,
                      ! requires a data copy
call sub( p )         ! p may not be contiguous,
                      ! requires a conditional copy
```

The third and fourth calls above must be copied more or less like the following, to make the actual argument contiguous:

```
real, allocatable :: tempb(:)
real, pointer :: tempp(:)
allocate( tempb(n/2) )
tempb = b(1:n:2)
call sub( tempb )
b(1:n:2) = tempb    ! required unless dummy
                    ! argument is intent(in)
if( is_contiguous( p(:) ) )then
   tempp => p
else
allocate( tempp(extent(p,1)) )
tempp = p
endif
call sub( tempp )
if( is_contiguous( p(:) ) )then
   ! nothing to do here
else
   ! not required if dummy is intent(in)
   p = tempp
   deallocate( tempp )
endif
```

This overhead is generally managed behind the scenes, with no hint from the compiler that the fourth call with the array pointer can be significantly more expensive than the second call with the allocatable array. A case like this can be alleviated by adding the Fortran 2008 contiguous attribute to the array pointer declaration:

```
real, dimension(:), pointer, contiguous :: p
...
call sub( p )          ! p now must be contiguous
                       ! no temp, no copy needed
```

You should be aware of these hidden overheads, which can be easily avoided unless you need the full flexibility of a true array pointer.

Stride-1

Fortran arrays are stored column-major; where in a two-dimensional array, indices are row and column, with consecutive elements down a single column (varying only the row index) laid out consecutively in memory. Another way to say this is that the column dimension is the high stride index. So `a(i,j)` is stored adjacent to `a(i+1,j)`. You should already be optimizing your loops so the stride-1 dimension is accessed in the inner loop, to take advantage of reference locality, which improves the cache hit rates:

```
do j = 1, n
do i = 1, n
   a(i,j) = ... ! a(i,j) is stride-1 in
                ! the inner loop
   b(j,i) = ... ! while b(j,i) is not
enddo
enddo
```

Fortran 90 added assumed-shape array arguments, where you don't have to specify the size of each dimension. Instead, each dimension assumes the size of the actual argument. The PGI compilers impose one additional constraint, that each assumed-shape dummy argument array preserve the stride-1 property in the leftmost dimension. This makes it easier to generate appropriate SIMD code for inner loops, assuming the loops are written as above to maximize stride-1 accesses in the inner loop.

However, this can also affect the overhead of procedure calls. Let's change the previous subroutine call example to use an assumed-shape array.

```
subroutine rout( x )
real :: x(:,:) ! assumed-shape dummy argument
               ! will be stride-1 in left dimension
...
end subroutine
...
real, dimension(100,200) :: a
real, dimension(:,:), allocatable :: b
real, dimension(:,:), pointer :: p
...
call rout( a )          ! a is contiguous and
                        ! leftmost-stride-1
call rout( a(:,2:50) )  ! section not contiguous
                        ! but still leftmost-stride-1
call rout( b )          ! b is allocatable contiguous
                        ! and leftmost-stride-1
call rout( b(:,1:n:2) )! section is not contiguous,
                        ! but still leftmost-stride-1
call rout( p )          ! p may not be contiguous and
                        ! may not be leftmost-stride-1
```

As with the assumed-size dummy argument, the compiler may have to allocate and fill a temporary array if the actual argument array or array section is not stride-1 in the leftmost dimension. It is hard to prove that an array pointer will always be stride-1 in the leftmost dimension, unless it has the contiguous attribute.

The other side effect of knowing that the leftmost dimension is stride-1 is that it simplifies the address code generation, even for single-dimensional arrays.

```
real, dimension(:,:), allocatable :: a
real, dimension(:,:), pointer :: p
...
do j = 1,n
do i = 1,n
   a(i,j) = p(i,j) ...
enddo
enddo
```

This loop accesses an allocatable array and an array pointer. In each case, a typical implementation will use an array descriptor that contains the address to the first word of the array, and then for each dimension contains the lower bound, the extent (or upper bound), and the stride for that dimension. Computing the offset for an array element **`p(i,j)`** is:

```
(i-lbound(p,1))*stride(p,1) + (j-lbound(p,2))*stride(p,2)
```

However, because **`a`** is an allocatable array, the compiler knows that **`stride(a,1)`** will always be exactly one, so the offset is:

```
(i-lbound(a,1)) + (j-lbound(a,2))*stride(a,2)
```

Recommendations

If you use array pointers but always allocate those arrays, you would probably be better served replacing the pointer attribute with allocatable. Even in those cases where you create other array pointers to point into that array, there are advantages to using allocatable, target instead of pointer. Pointer arrays have all the performance limitations mentioned above. Arrays with the allocatable, target attributes will only have the aliasing limitation, and then only with array pointers; two allocatable, target arrays cannot alias each other, and the compiler can use that information.

For those cases where you really need to use array pointers, apply the Fortran 2008 contiguous attribute whenever possible. This will give the compiler more information and enable it to generate more efficient code.

Since Fortran 90, derived type members can also have the pointer attribute, and since Fortran 95, derived type members can have the allocatable attribute:

```
type pre_mat
   real, dimension(:), allocatable :: m1
   real, dimension(:), pointer :: p2
end type
```

Many programs written after Fortran 90 was introduced and implemented, but before Fortran 95 compilers were available, use the pointer attribute for derived type members. If your program uses the pointer attribute in derived types when those members are always allocated, you will get better results if you use the allocatable attribute instead.

In summary, prefer allocatable to pointer if the array pointer never appears in a pointer assignment. For an array pointer, if the target is always contiguous, add the contiguous attribute. These guidelines will lead to the generation of more efficient code for your Fortran programs.

PGI®