The PGI Fortran and C99 OpenACC Compilers

Brent Leback, Michael Wolfe, and Douglas Miles
The Portland Group (PGI)
Portland, Oregon, U.S.A
brent.leback@pgroup.com

Abstract—**This paper provides an introduction to programming accelerators using the PGI OpenACC implementation in Fortran and C, which is based on OpenACC API version 1.0. The paper explains the use of the data construct, and compares the use of the Parallel and Kernels construct. PGI-specific extensions and features, and compiler and runtime options, are shown.**

Keywords-OpenACC, accelerator, GPGPU, programming models

I. INTRODUCTION

In the second half of 2011, a group of vendors came together with a goal to standardize a set of directives for programming accelerators. The OpenACC Application Program Interface (OpenACC API) is the result of these meetings. OpenACC allows programmers to write applications that offload work (both code and the associated runtime data the code operates on) from a host CPU to an attached accelerator, typically a GPU or similar device. Unlike explicit languages such as CUDA or OpenCL, details of data transfer between the host and device, kernel launch configurations, and synchronization, are all virtualized in the programming model.

Portions of the OpenACC API borrow heavily from PGI's Accelerator Programming Model. PGI introduced its Accelerator Programming Model in 2008 and it has undergone a couple of revisions since that time. The goal all along has been to provide a higher-level model aimed at scientists, engineers, and maintainers of large legacy codes. The model itself is directive based, in the spirit of OpenMP, as opposed to the CUDA or OpenCL languages which require adherence to extensive runtime APIs and usually a complete overhaul and rewrite of the low-level kernel functions. In a directive-based model, programmers insert comments in the form of directives in Fortran, and pragmas in C and C++, which denote to the compiler which data structures need to be copied to the device, and which blocks of code should run on the device. Ideally, the code is left in a state that, while perhaps not pristine, will work as well as the original when run through a compiler that does not recognize the accelerator directives.

As the PGI Model began to gain traction among members of the scientific and engineering communities, other software development tool vendors provided solutions which were, in many of the functional areas, similar in scope, but they contained partial or even sometimes extended functionality; at the very least the directives were spelled differently. In 2010, PGI began an effort within the OpenMP committee to standardize a directive model. Members involved with the effort studied many options, and it appeared that resolving them would take much longer than initially thought. At CUG in 2011, this author stated that it certainly appeared that it was in danger of dying. Eventually, this state of affairs led to a series of separate meetings between Cray, NVIDIA, and PGI outside of the OpenMP umbrella. CAPS later joined in, and by Supercomputing 2011 the OpenACC 1.0 Specification was complete. Today, implementations by the vendors are well underway, and there appears to be a chance the specification, or something similar, will be pulled back into the OpenMP domain at some point in the future.

II. PROGRAMMING ACCELERATORS USING DIRECTIVES

A. *The Host Side*

All host programs which make use of an accelerator must perform the following steps:

1. Select the attached accelerator; initialize the device and the runtime
2. Allocate data on the device
3. Move data from the host, or initialize it there on the device
4. Launch a kernel or series of kernels
5. Gather results back from the device
6. Deallocate the data, free the device

CUDA C/C++ is an explicit language for programming accelerators. The steps are clear, all details are left to the programmer, and it leaves little room for confusion or doubt. Typical CUDA host code for performing the above steps looks like this:

```
/* Step 1, showing default behavior */
cudaSetDevice( 0 );


/* Step 2 */
msize = sizeof(float)*n;
cudaMalloc(&da,msize);
cudaMalloc( &db, msize );
cudaMalloc( &dc, msize );
```

```
/* Step 3 */
cudaMemcpy( db, b, msize, cudaMemcpyHostToDevice );
cudaMemcpy( dc, c, msize, cudaMemcpyHostToDevice );

/* Step 4 */
dim3 threads( 256 );
dim3 blocks( n/256 );
vaddkernel<<<blocks,threads>>>( da, db, dc, n );

/* Step 5 */
cudaMemcpy( a, da, msize, cudaMemcpyDeviceToHost );

/* Step 6 */
cudaFree( da );
cudaFree( db );
cudaFree( dc );
```

Even CUDA Fortran, which uses higher level syntax than CUDA C, is explicit in following these steps, and could be considered even more so since da, db, and dc are declared differently than a, b, and c in that they have the device attribute, and are therefore limited in where they can be used in the language:

```
real, device, dimension(:), allocatable :: da, db, dc

 allocate( da(1:n), db(1:n), dc(1:n) )   ! step 2

db = b   ! step 3
dc = c

call vaddkernel<<<n/256,256>>>( da, db, dc, n )   ! step 4

a = da   ! step 5

deallocate( da, db, dc )   ! step 6
```

In an implicit, directive-based model, the details of the five or six steps which all accelerated host programs must have are implied or understood by the compiler based on the kernel work that is specified, and as we'll see later, by the surrounding context of the program unit. Take, for instance, this annotated loop:

```
#pragma acc kernels loop
for (i = 0; i < n; i++)
    a[i] = b[i] + c[i];
```

Through compiler analysis it can be determined that data areas for a, b, and c of length n must be allocated on the device, that data for b and c must be copied from the host to the device, that the results in a must be copied from the device back to the host, and with a lack of any other surrounding context, space for all three arrays should be freed.

Of course in the real world, code is much more complicated. Arrays are multi-dimensional, accesses into the arrays are not sequential, perhaps not even regular, complex data structures are used, there are function calls, etc. Much of the work over the last several years on the PGI Accelerator Model has gone into addressing these issues while trying to stay true to the original design goal of handling most of the target-specific details within the compiler, deciding which parameters should be controllable through directive clauses, and providing feedback which allows users to understand the decisions the compiler made and why.

B. *The Device Side*

While most traditional compiler users understand assembly language, they don't want to drop down to that level in the normal course of their day. The same is true of many GPGPU programmers: they may have written some CUDA or OpenCL as they were ramping up the learning curve, but they would prefer to leave the device code generation to either someone else, such as their younger colleagues or a library vendor, or something else, preferably their high-level tools. To understand the challenges in providing robust tools, let's first look at the typical hardware that these tools target.

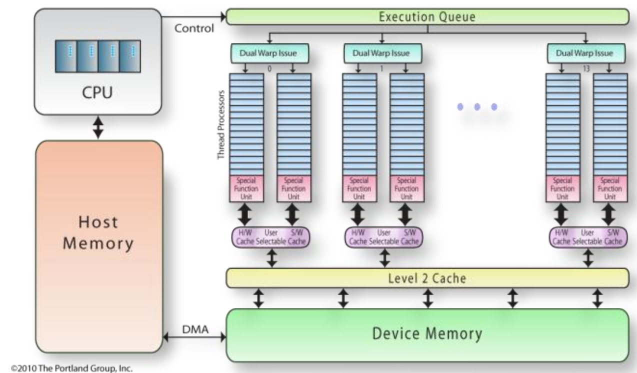

Figure 1. NVIDIA Fermi block diagram. In this instance, fourteen streaming multiprocessors each containing thirty-two thread processors.

Most current GPUs support at least two levels of parallelism: an outer doall (fully parallel) loop level, and an inner synchronous (SIMD or vector) loop level. Each level can be multidimensional with 2 or 3 dimensions, but the domain must be rectangular. The outer doall level, which in OpenACC is called gangs, is typically mapped to the processing elements commonly referred to on NVIDIA GPUs as the streaming multiprocessors, or in CUDA as a thread block. Likewise, the number of gangs, in CUDA parlance, is the number of blocks in a grid. No synchronization is supported by the programming model between parallel threads across the gang level. This is an important point. If you are porting an existing region of host code to a GPU, and you suddenly find a reference to something that you think will be computed in another gang (or the compiler informs you of such), you must either work to remap the code using a different schedule or break the

code into multiple kernels, since the order of execution of kernels is deterministic.

At the inner SIMD or vector level, explicit synchronization is supported and required. In OpenACC, this is termed the vector level. This level is familiar to HPC programmers: it can be thought of as equivalent to SIMD registers on an x86. All fully parallel loops, with no dependencies between iterations, can be scheduled for either doall or synchronous parallel execution, but by definition SIMD vector loops that require synchronization can only be scheduled for synchronous parallel execution.

In addition, OpenACC supports an additional level of parallelism, corresponding to multiple threads that interleave execution on a single core. This is called the worker level; it can be considered similar to Intel hyperthreading or SPARC multithreading. Loop iterations that share data or synchronize can be run in parallel at the worker level, since those workers will execute on the same physical core, in the same gang, and share those resources.

III. Introducing OpenACC

As mentioned above, OpenACC is a new parallel programming standard describing a set of compiler directives which can be applied to standard C, C++, and Fortran to specify regions of code and associated data for offloading from a host CPU to an attached accelerator. At the most-basic level, by merely annotating compute-intensive regions of code with directives, the programmer can target an accelerator with his or her existing program.

After running a few simple loops and timing the results, however, it will become clear that care must be taken to control and optimize data locality. A naive approach that copies a kernel's data back and forth at each kernel invocation will almost always run slower than simply leaving the computation on the CPU. The primary mechanism to control data locality in OpenACC is the Data Construct.

A. *Data Construct*

The data construct defines a region where arrays, subarrays, and scalars should be allocated on the device. Clauses in the data construct further define whether data should be copied from the host to device upon region entry, copied back to the host as the program exits the region, or just created and used entirely on the device.

The syntax of the data construct in C and C++ is:

```
#pragma acc data [ clause [[,] clause ] ]
{
}
```

And in Fortran:

```
!$acc data [ clause [[,] clause ] ]
!$acc end data
```

Commonly used clauses in the data construct are:

if (condition) -- Create the device copies if the condition is true

copy (list) -- Create and copy host data both in and out
copyin (list) -- Create and copy host data in only
copyout (list) -- Create and copy device data out only at region exit
create (list) -- Create the device data structure but no copying
deviceptr (list) -- This pointer or dummy argument references device memory already. Don't create or copy.

Many programs need only one data construct in the entire program. It usually goes after the corresponding host arrays have been created and initialized using i/o, which of course is only supported on the host. Here is an example from an OpenACC version of the swim benchmark:

```
C     INITIALIZE CONSTANTS AND ARRAYS
C
      CALL ALLOC

!$ACC DATA CREATE(U(NP1,MP1), V(NP1,MP1))
!$ACC& CREATE(UNEW(NP1,MP1),VNEW(NP1,MP1))
!$ACC& CREATE(PNEW(NP1,MP1), UOLD(NP1,MP1))
!$ACC& CREATE(VOLD(NP1,MP1), POLD(NP1,MP1))
!$ACC& CREATE(CU(NP1,MP1), CV(NP1,MP1))
!$ACC& CREATE( P(NP1,MP1),Z(NP1,MP1))
!$ACC& CREATE(H(NP1,MP1), PSI(NP1,MP1))

      CALL INITAL
```

Two other data directives are commonly used. A declare directive is similar to the data directive, but uses the scope of the entire function, subroutine, or program as an implicit data region. It is inserted after the declaration, and takes most of the same clauses as the data directive.

The other is an executable directive and is used to synchronize the host and device copies of the array, subarray, or scalar while you are inside of a data region, either explicit or implicit. It's syntax in C/C++ is:

```
#pragma acc update [ clause [[,] clause ] ]
```

And in Fortran:

```
!$acc update [ clause [[,] clause ] ]
```

and the common clauses are:
if (condition) -- Do the update if the condition is true
device (list) -- Update the device copy from the host data
host (list) -- Update the host copy from the device data

At least one instance of device or host must appear in the directive. The update directive is commonly used in a number of situations:

1. When sharing computation between the host and device.

2. Gathering some set of intermediate results from the device to perform a conditional test on the host.
3. Writing intermediate results generated on the device using i/o which is only supported on the host.
4. Collecting halos of device data for sharing with other processes via MPI or some other communication mechanism.
5. Storing halos or new parameter values collected from other processes or host inputs back to the device.

In practice, we've found that collecting halos into one contiguous buffer and doing one update, packing and unpacking on both sides, usually performs better than doing multiple updates.

B. *Present Data*

One change between the PGI Accelerator Model and OpenACC should potentially help developers when porting codes to OpenACC. When accelerator compute regions are not lexically within data regions, i.e. there has been a subprogram call, the PGI Accelerator Model requires the reflected attribute for the procedure argument. Then at compile time the compiler will insert a hidden argument into the calling sequence so both the host pointer and device pointer can be passed along. OpenACC uses a runtime lookup table, keyed by the host address, to determine if a device copy of the data exists, and if so, fetches the appropriate device address from this table. Thus far we've found the overhead for the lookup, which exists entirely in host memory, is minimal.

Programmers can explicitly list subprogram variable names using these clauses on the data or declare directives and direct the runtime behavior:
present (list) -- Device copy must be present
present_or_copy (list) -- If device copy not present, create and copy
present_or_copyin (list) -- If device copy not present, create and copyin
present_or_copyout (list) -- If device copy not present, create and copyout
present_or_create (list) -- If device copy not present, just create

The OpenACC specification states, and the PGI compiler has defaults, which make the use of these present clauses unnecessary in many cases. For instance, arrays which appear inside the kernels construct which do not appear in any enclosing data clauses are treated as present_or_copy. It is also worth noting that these clause names can be shortened, for example, pcopyin is synonymous with present_or_copyin.

C. *Kernels Construct*

The OpenACC kernels construct has been supported since PGI 12.3, and is a straight-forward translation from the PGI Accelerator region construct. It specifies a region of code that is to be compiled into one or more accelerator kernels, executed in sequence. The kernels directive itself can take most of the data clauses presented previously. Each kernel can differ in the number of gangs and workers it utilizes. For instance, simple reductions can be recognized and implemented with two kernels, one for local reductions within a gang, and one to do the final reduction using the results generated by the first kernel.

Earlier, we presented a combined directive that contained both a kernel construct and a loop construct. For more complicated cases, separate loop constructs are usually inserted inside of the kernels construct. Within a kernels construct, these loop constructs are typically used to describe what type of parallelism to use to execute the loop.

Clauses which are typically used on the loop directive when they are within the kernels construct are:
collapse (n) -- Collapse n tightly nested loop iterations
gang (n) -- Iterations of loop executed across n gangs
worker (n) -- Iterations of loop executed across n workers per gang
vector (n) -- Iterations of loop executed in strips of length n
seq -- Iterations of loop executed sequentially within a worker
independent -- Assert that iterations of loop are data-indepedent to each other

Again from the swim benchmark, here is an example of loop directives contained with the kernels construct:

```
!$ACC KERNELS
!$ACC LOOP GANG, VECTOR(16)
DO 100 J=1,N
!$ACC LOOP GANG, VECTOR(16)
     DO 100 I=1,M
          CU(I+1,J) = .5D0*(P(I+1,J)+P(I,J))*U(I+1,J)
          CV(I,J+1) = .5D0*(P(I,J+1)+P(I,J))*V(I,J+1)
          Z(I+1,J+1) = (FSDX*(V(I+1,J+1) –
    1                    V(I,J+1))-FSDY*(U(I+1,J+1)
    1     -U(I+1,J)))/(P(I,J)+P(I+1,J)+P(I+1,J+1)+P(I,J+1))
          H(I,J) = P(I,J)+.25D0*(U(I+1,J)*U(I+1,J)+U(I,J )*
    1        U(I,J) + V(I,J+1)*V(I,J+1)+V(I,J)*V(I,J))
100 CONTINUE
```

As this example shows, one of the advantages of the kernels construct over the parallel construct is the flexibility the compiler or programmer has in scheduling the loops using multi-dimensional decompositions, which can enable tiling and data-reuse using shared memory.

Here is a listing of the compiler -Minfo output for this loop:

```
162, Loop is parallelizable
164, Loop is parallelizable
 Accelerator kernel generated
162, !$acc loop gang, vector(16) ! blockidx%y threadidx%y
164, !$acc loop gang, vector(16) ! blockidx%x threadidx%x
        Cached references to size [17x17] block of 'p'
        Cached references to size [17x17] block of 'v'
        Cached references to size [17x17] block of 'u'
```

The compiler is able to generate device code that utilizes 17x17 blocks of shared memory to hold a tile of p, u, and v for each gang.

One current drawback of the kernels construct is that loops need to be tightly nested for the compiler to be able to generate good kernels. At PGI, we hope to address that limitation as we gain experience with the parallel construct.Define abbreviations and acronyms the first time they are used in the text, even after they have been defined in the abstract. Abbreviations such as IEEE, SI, MKS, CGS, sc, dc, and rms do not have to be defined. Do not use abbreviations in the title or heads unless they are unavoidable.

D. Parallel Construct

The OpenACC API also defines another type of compute construct, the parallel construct. The parallel construct is more explicit user-specified parallelism than the kernels construct, and much like the OpenMP parallel construct. The OpenACC parallel construct immediately launches a specified number of gangs, where each gang contains a specified number of workers. One worker in each gang starts executing the code in the parallel construct redundantly, just like threads in an OpenMP parallel construct, until they reach a work-sharing loop construct. At that point, the iterations of the work-sharing loop are spread across the gangs and workers within each gang, as specified by the construct. By default, outer loops are shared across the gangs. Inner loops are work-shared amongst the workers in a gang, a.k.a. the threads in the threadblock.

The parallel construct results, in many cases, in the annotated OpenACC loop to look very similar to the exising OpenMP annotations over the same structure. This is an advantage to this construct that may make initial ports to OpenACC easier.

In OpenMP, the number of threads is fixed once you enter a parallel region, except in cases of nested parallelism. Likewise, with the parallel OpenACC construct, the number of gangs, and the number of workers in a gang, is fixed upon entering the parallel region. Only one kernel will be generated and launched at runtime.

Here is a snippet from a Fortran nbody example that shows the use of the parallel construct:

```
!$acc parallel
!  gang-redundant work goes here
!$acc loop gang
do j = 1, n
     p1 = posin(j,1)
     p2 = posin(j,2)
     p3 = posin(j,3)
     f1 = 0.0; f2 = 0.0; f3 = 0.0
     !$acc loop worker, reduction(+:f0,f1,f2)
     do i = 1, n
       r1 = posin(i,1) - p1
       r2 = posin(i,2) - p2
       r3 = posin(i,3) - p3
       distsqr = r1*r1 + r2*r2 + r3*r3
       distsqr = distsqr + softensqr
       distinv = 1.0 / distsqr
       distinv3 = distinv*distinv*distinv
       s = posin(i,4) * distinv3
       f1 = f1 + r1*s
       f2 = f2 + r2*s
       f3 = f3 + r3*s
     end do
     fmassinv = posin(j,4)
     v1 = velocity(j,1) + f1*fmassinv*dtime
     v2 = velocity(j,2) + f2*fmassinv*dtime
     v3 = velocity(j,3) + f3*fmassinv*dtime
     . . .
   end do
   !$acc end parallel
```

Clauses available in the Parallel construct and not in the Kernels construct:

num_gangs() - fixed number of gangs in the parallel construct

num_workers() - fixed number of workers per gang used when a worker-shared parallel loop is encountered

reduction() - explicitly defined reduction operator, similar to OpenMP

vector_length() - defines the vector length for SIMD operations in the loop

private() - data that is replicated, one copy per gang

firstprivate() - replicated like private, but initialized with the host values

PGI support for the parallel construct will be first enabled in PGI 12.5 and available in May, 2012.

IV. INTEROPERABILITY AND PGI-SPECIFIC FEATURES

The PGI Accelerator compilers will accept code containing both OpenACC directives and PGI Accelerator Model directives. Generally, it will be best to recompile all code with the same version of PGI compiler to guarantee data directives are handled consistently between the two models.

Besides renaming the directives, users looking to migrate completely from the PGI Accelerator Model to OpenACC need to be aware of a few other changes:

1. C Subarrays are specified differently in OpenACC. Since the C language has no standard subarray notation, PGI used notation

that matched Fortran, with lower and upper bounds. OpenACC has adopted Intel's Array Notation for C, which uses a starting index and length. So, x[0:n-1] for n elements of x has become x[0:n].

2. OpenACC only allows contiguous subarrays to be allocated on the device.
3. Reductions are explicit in the parallel construct.
4. OpenACC uses runtime present checks rather than reflected as mentioned above. There is no support for mirrored data in OpenACC.
5. PGI may accept declaration-style array specifications in data clauses when in fact they should be array or subarray sections.

CUDA Fortran and OpenACC can also coexist. As an extension, PGI OpenACC can properly recognize and operate with CUDA Fortran device arrays. OpenACC kernels can also call CUDA Fortran device functions.

A. Compiler Options, Feedback, and Profiling

We are currently using compiler options from the PGI Accelerator Model to control GPU code generation. With the first 1.0 general release of the OpenACC functionality we will determine whether or not we need to pull these into the OpenACC specific flags. Until then, these are the pertinent flags for users to be aware of:

-acc Enable recognition of OpenACC directives
-acc=strict|verystrict Level of warnings for non-OpenACC directive conformance
-Minfo=accel Compiler will produce informational messages about accelerator optimization
-ta=nvidia Control NVIDIA target code generation options

We have shown some examples of the Minfo accelerator compiler output above. The key things to look for are the amount of data transfered for each array, and the loop schedules generated for each accelerator kernel.

Runtime profiling can be controlled in the PGI OpenACC compiler, and in the PGI accelerator model, by use of the PGI_ACC_TIME environment variable. Setting this variable will instruct the runtime to print launch statics for each kernel that was generated, and data region encountered. The statistics include the number of launches, the launch configuration, and the time is microseconds (total, max, min, and average) for each kernel. If data transfer is required for the kernel, that is included as well. Otherwise, for data regions, the output contains total time for data transfer for that region.

Accelerator Kernel Timing data for a simple example:

```
C:\Users\Leback\openacc\samples\acc_f3a.f90
 smooth
  32: region entered 20 times
    time(us): init=0
    34: kernel launched 20 times
      grid: [128x128] block: [16x16]
      time(us): total=221085 max=11059 min=11051
avg=11054
C:\Users\Leback\openacc\samples\acc_f3a.f90
 smooth
  24: region entered 20 times
    time(us): init=0
    26: kernel launched 20 times
      grid: [128x128] block: [16x16]
      time(us): total=221083 max=11058 min=11050
avg=11054
C:\Users\Leback\openacc\samples\acc_f3a.f90
 main
  128: region entered 1 time
    time(us): init=0
         data=43765
```

For data other than time-based information, PGI utilizes its own accel_lib an f90 module, and a corresponding C library to extract the number of data transfers and number of regions entered. Users can insert these calls into their code to get counter values that are kept at runtime, from either C or Fortran.

```
integer function acc_regions()
end function
integer function acc_kernels()
end function
integer function acc_allocs()
end function
integer function acc_frees()
end function
integer function acc_copyins()
end function
integer function acc_copyouts()
end function
integer(8) function acc_bytesalloc()
end function
integer(8) function acc_bytesin()
end function
integer(8) function acc_bytesout()
end function
```

V. CONCLUSION

In this paper we've touched on what we believe to be the most commonly used OpenACC v1.0 features. For a complete list of features, OpenACC users are encouraged to download the entire OpenACC specification from www.openacc.org. Areas we have not touched on in this paper include asynchronous control, the OpenACC runtime library, and environment variables. We fully expect the API specification to change as developers gain more experience and as the baseline for accelerator target architectures gain added functionality.

REFERENCES

[1] The OpenACC Application Programming Interface, Version 1.0, November 2011. www.openacc.org

[2] Michael Wolfe, High Performance Compilers for Parallel Computers, Addison-Wesley, 1996

[3] The PGI Accelerator Compilers with OpenACC, by Michael Wolfe, PGI Compiler Engineer. http://www.pgroup.com/lit/articles/insider/v4n1a1.htm