Accelerating codes with OpenACC directives and the Cray Programming Environment on the Cray XK7

OpenACC is a directive-based method for porting existing Fortran, C and C++ codes to execute on one or more accelerators. By way of worked example, this document describes the process of porting the MultiGrid (MG) code from the NAS Parallel Benchmark suite to run on a GPU using OpenACC directives in the original source code. The development environment is the Cray XK7 and, in the process, we show how the Cray Programming Environment (PE) tools make the process much easier.

The MG code is a good example. It is much shorter than a typical scientific or engineering application, but it does have a similar structure. We concentrate on the serial version of the benchmark, but the same process has been used to develop an OpenACC port of the MPI version of the benchmark. This tutorial covers the official NAS Fortran version of the benchmark[1].

This document does not introduce the OpenACC directives. The OpenACC standard[2] can provide this, as can

```
man intro_openacc
```

when the Cray accelerator module is loaded (as described below).

How to use this document

This document can be used in two ways. The more productive is to work through the document logged in to a Cray XK7, examining the supplied code versions (or attempting the steps yourself). You can then compile and run the code and obtain the results for yourself. This can be done as part of an organised course or as a standalone exercise.

We also give sample output and results, so you can also read this document and associated source code without access to a Cray XK7 for compilation and running.

Commands and source are shown `like this`. Sample output is show `like this`. The quoted results are for a Cray XK7. As the Cray software stack is continually improved, all the performance figures in this document should be viewed as indicative.

Be careful if you cut-and-paste commands out of this document: what appear to be simple ASCII hyphens may actually be more complicated symbols in the PDF. If a pasted command doesn't work, try retyping the hyphens.

Finally, this document contains many results obtained using pre-release software, which are liable to change and should only be regarded as indicative.

For the CSCS course today, this document will lead you through Practical3.

Alistair Hart (ahart@cray.com), 18.Feb.13.

[1] http://www.nas.nasa.gov/publications/npb.html.
[2] Available from http://www.openacc.org.

1. Before you start

The Cray Compilation Environment (CCE) is part of the larger Cray Programming Environment (PE). We will use this to compile the OpenACC directives, so you should first make sure you know its features and check your code compiles and executes correctly. If you already use the Cray PE, you will find much (but maybe not all) familiar.

1.1. Cray Programming Environment and compiler familiarisation

Most software on the Cray XK7 is packaged as modules. For OpenACC programming, you will need to use either the Cray or PGI Programming Environment. Check that either modules `PrgEnv-cray` or `PrgEnv-pgi` are loaded. These should automatically load the associated compilers (`cce` and `pgi`, respectively). You can see which modules are loaded by typing:

```
module list
```

They are usually loaded by default on the Cray XK7. You should check that the most up-to-date version of the compiler is being used by typing, for instance,

```
module avail cce
```

and then swapping modules if needed. For OpenACC to be supported (rather than directives ignored as comments), you also need to load the additional Cray accelerator module (which may not be loaded by default, check by listing):

```
module load craype-accel-nvidia35
```

For these exercises, we provide a bash script that can be used to switch programming environments and select the correct compiler. The script should be sourced in the current shell, rather than executed in a new one. As argument it can optionally take `cray` or `pgi`; without argument it defaults to `cray`. You can find this in the top-level `Tools` directory in the supplied exercises. Use it like this before you start compiling:

```
source Tools/XK_setup.bash
```

1.1.1. The Cray Compilation environment

The CCE compilers are named `crayftn` for Fortran and `craycc` for C (the C++ compiler `crayCC` is not covered here). There are `man` pages for each of these[3]. These commands should not be used directly. Instead, commands `ftn` and `cc` (or `CC` for C++) should be used. These are wrappers that add appropriate paths and low-level options. This means you should not normally need to add explicit `-L` or `-l` options for the Cray-provided libraries. Other compiler flags are passed down to the CCE compilers. With the CCE, the default optimisation level is `-O2`, and this should be used in most cases. If you require unoptimised code, you must explicitly specify `-O0`.

OpenMP support is enabled by default. If OpenMP is not required, use the flag `-hnoomp`. If module `craype-accel-nvidia35` is loaded but OpenACC is not required, it can be disabled using `-hnoacc`. Note that this will not disable all accelerator-based functionality, so in some cases the module should be unloaded before compiling.

Compiler commentary is available in an annotated “loopmark” listing of the file, giving further information on, for instance, loop transformations and optimisations and, most important here, OpenACC transformations. Loopmark feedback is requested by compiler flag `-hlist=a`, and is written to files whose name has the same stem as the source file and extension `.lst`. This loopmark information is particularly valuable for OpenACC kernels. For a more restricted set of messages, replace `a` by `m`. For even lower-level information (e.g. to identify pattern-matched routines or to understand OpenACC synchronisation points), compile with `-hlist=d` (or `-hlist=ad` for loopmark

[3] Full documentation is available from http://docs.cray.com.

as well). Two compiler-generated files are produced with extensions `.opt` (from the optimiser) and `.cg` (from the (lower-level) code generator). These are harder to read, but include API calls inserted by the compiler.

Compiler messages are prefixed by an identifier (a string and a number, e.g. `ftn-6405`). More information on these messages can be obtained using the command

```
explain <identifier>
```

It is usually also a good idea to compile and run the code with array bounds checking activated on the CPU and to fix any problems identified before starting with OpenACC. The compiler options are `-Rbcps -O0` for Fortran and `-hbounds -O0` for C. Not that optimisation should be switched off for full bounds checking. Third party memory checkers like `valgrind` can also be used, especially for C codes with dynamic memory allocation where bounds checking is less useful.

1.1.2. Other OpenACC compilers

The PGI compiler are `pgfortran`, `pgcc` and `pgCC`, for which there are man pages (accessible when `PrgEnv-pgi` is loaded). Again, you should use the wrapper functions `ftn`, `cc` and `CC` in place of these. Although these exercises are based around the Cray compiler, the basic command for compiling code is:

```
ftn -ta=nvidia -Minfo=accel
```

The CAPS compilers, if available, are not packaged as a PrgEnv module and will not use the wrappers.

2. Overview of the code

The code example used here is the single-process MG Multigrid Code from the NAS Parallel Benchmark suite. There is no parallel domain decomposition using MPI or similar. The code does contain OpenMP directives for shared memory parallelism but we will not use them.

The code can be found in directory `Practical3` of the tutorial materials; you can choose to use either the Fortran or C versions, which are equivalent.

The code is compiled from the top-level directory (`Practical3/F` or `Practical3/C`) using the command:

```
make MG [CLASS=<CLASS>] [<OPTIONS>]
```

Option `CLASS=<CLASS>` selects one of various hardwired problem sizes. Class `B` (the default) is the largest that runs on a single GPU, and we will use this for all the examples here[4]. There are various optional flags, which we will describe later.

The options are passed to the `Makefile` in subdirectory `MG`, where the source code also lives. The options are converted into CCE-specific flags in `config/make.def`. We give various versions of the source code, illustrating the steps described below. Version `00` (the default) is the original code as downloaded. To select other versions, e.g. `03`, use option `VERSION=03`. Command: `make clean` (or: `make veryclean`) should be used when switching versions.

This automated building of the code is used for convenience. The exact compiler commands are echoed to STDOUT. The executables are created in directory `bin` and have name `mg.<CLASS>.x`.

The code can be compiled using:

```
make MG
```

[4] Class `A` is the same problem size as Class `B` but runs for fewer iterations. Classes `S` and `W` are much smaller and suitable only for functionality and correctness testing.

To run the code using the PBS batch system, you can use the supplied script to write and submit a PBS batch script using the lustre filesystem:

```
bash submit.bash bin/mg.B.x
```

The script will report the directory it creates on the screen, where the output will be found in the log file.

Once run, there are three important lines in the output. Here are some sample results for Fortran:

```
L2 Norm is   0.1800564401355E-05
Mop/s total     =                    1623.04     ! Fortran
Verification    =                 SUCCESSFUL
```

The code should verify that the answer is correct (also check that the Norm is not a NaN). The Mop/s performance figure is calculated by the code (rather than with hardware counters)[5]. Clearly the C version is still slightly less efficient, but our goal here is to port to OpenACC.

3. Step 1. Knowing where to start: profiling the code

The first step is to profile the code running on CPU cores to: understand the structure (call-tree) of the code; discover which routines (and which loopnests within them) take the most time; find the typical tripcounts of these loops; and see how data is moved between the subprograms (subroutines or functions). Whilst this is trivial for simple codes, it is a significant task for even moderately complicated codes like MG. The Cray PE offers a number of useful tools to help with this.

3.1. Profiling the code with the Cray Performance Analysis Tools (CrayPAT)

CrayPAT is a powerful performance analysis tool in the Cray PE, provided in the `perftools` module. Profiling a code is usually a two-stage process. First a "sampling experiment" is run, which profiles the code using a statistical analysis. Based on this, a second profile can be generated using a "tracing experiment" which is more accurate. Using the "Automatic Program Analysis" (APA) feature of CrayPAT, the results of a sampling experiment are used to minimise the overhead of the tracing experiment by concentrating on the important subprograms. For more details, run the commands:

```
man intro_craypat
man pat_build
pat_help
```

As the MG code is relatively small, we can simplify this process and go straight to a tracing experiment. We use `VERSION=01` with an optional feature; the CrayPAT API allows us to insert calls in the code that restrict the profiling to just the iteration loop (the `CRAYPAT` preprocessing macro is automatically set by the `perftools` module).

The procedure is as follows:

1. Load the module:
   ```
   module load perftools
   ```
2. Carry out a clean build of the code, building, for instance, executable `mg.B.x`:

   ```
   make clean
   make MG VERSION=01
   ```

[5] This performance figure is estimated within the code using hardwired factors. It is not based on CPU hardware counters. We will use the same measure for the GPU; there are currently no NVIDIA hardware counters for GPU op/s.

Note that you should always profile a representative scale problem, not a simple test case.

3. Instrument the code, tracing all subprograms, and creating executable mg.B.x+pat:

   ```
   cd bin
   pat_build -f -u mg.B.x
   cd ..
   ```

   The -f flag allows pat_build to overwrite file mg.B.x+pat if it exists.

4. Run the instrumented executable

   ```
   bash submit.bash bin/mg.B.x+pat
   ```

5. This should generate a CrayPAT data file with extension .xf. Process this data file:

   ```
   pat_report <xf file>
   ```

   CrayPAT can generate a wide variety of reports (see: man pat_report for details). Running pat_report produces a file with extension ap2. You can explore this graphically using the apprentice2 application included in the perftools module (although transatlantic bandwidth may make this impossible):

   ```
   app2 <ap2 file>
   ```

3.2. Sample CrayPAT reports

Here we give a sample CrayPAT report. This is for the Fortran version of the code.

```
Table 1:  Profile by Function Group and Function

 Time%   |      Time   | Imb.   |  Imb.   | Calls   |Group
         |             | Time   | Time%   |         | Function

 100.0% | 12.069520 |     -- |      -- | 1630.0 |Total
|----------------------------------------------------------
| 100.0% | 12.069417 |     -- |      -- | 1230.0 |USER
||---------------------------------------------------------
||  54.9% |  6.620529 |     -- |      -- |  161.0 |resid_
||  25.3% |  3.057070 |     -- |      -- |  160.0 |psinv_
||   9.5% |  1.148982 |     -- |      -- |  140.0 |rprj3_
||   8.1% |  0.983395 |     -- |      -- |  140.0 |interp_
||   1.3% |  0.153775 |     -- |      -- |  461.0 |comm3_
|==========================================================
```

It is clear that four routines dominate the runtime. We can learn more about the structure of the code by generating the calltree:

```
pat_report -O calltree <xf file>
```

which gives

```
Table 1:  Function Calltree View

 Time%   |      Time   | Calls   |Calltree

 100.0% | 12.069520 | 1630.0 |Total
|---------------------------------------
| 100.0% | 12.069417 | 1230.0 |mg_
||--------------------------------------
||  72.3% |  8.724588 | 1180.0 |mg3p_
|||-------------------------------------
```

```
3||  28.3% |  3.416675 |   280.0 |resid_
3||  25.8% |  3.108020 |   320.0 |psinv_
3||   9.6% |  1.160157 |   280.0 |rprj3_
3||   8.1% |  0.983395 |   140.0 |interp_
|||====================================
||  27.3% |  3.295504 |    42.0 |resid_
|======================================
```

This helps in planning the movement of the data to and from the GPU. It is, however, a simplified view that omits routines that take a negligible amount of time, but which we will need to consider when porting. A full calltree can be generated by adding flag `-T` to the `pat_report` command; you will then see where `norm2u3`, `zero3` and `comm3` appear in the calltree.

3.3. Profiling with loop statistics

To assess suitability of loopnests for porting to GPUs, we need information on inclusive time spent in the loopnests and the typical tripcount of the loops. CrayPAT can generate this. The code should be compiled using flag `-hprofile_generate`. More details can be found in the `crayftn` and `craycc` man pages.

The code can be built using command:

```
make clean
make MG VERSION=01 PROFILE_GENERATE=yes
```

Once recompiled, the code is instrumented for CrayPAT and run as above. The loop profiling information is then generated by reprocessing the final data file as follows.

3.3.1. CrayPAT loop statistics information

Inclusive loop times are generated by using `pat_report` as before. Here are the lines relevant to the most expensive `resid` routine:

```
Table 2:  Loop Stats by Function (from -hprofile_generate)

    Loop   |   Loop   |   Loop   |  Loop   |  Loop   |Function=/.LOOP[.]
    Incl   |    Hit   |  Trips   | Trips   | Trips   |
    Time   |          |    Avg   |   Min   |   Max   |
   Total   |          |          |         |         |
|-----------------------------------------------------------------------
| 6.830878 |      161 |  96.497 |       4 |     256 |resid_.LOOP.1.li.634
| 6.830032 |    15536 | 201.067 |       4 |     256 |resid_.LOOP.2.li.635
| 4.033780 | 3123776 | 237.548 |       6 |     258 |resid_.LOOP.3.li.636
| 2.607888 | 3123776 | 235.548 |       4 |     256 |resid_.LOOP.4.li.642
```

3.4. Variable scoping with Reveal

The profiles make clear that the OpenACC work should start in the `resid` subroutine. As with OpenMP, to use the OpenACC directives we need to understand the scoping of the variables. That is, we need to understand whether variables are private, shared or otherwise.

Here this is simple but in more complicated cases, we can use the Cray Reveal tool to help.

Version `00` should then be recompiled, with OpenMP disabled:

```
make veryclean
make MG PROGRAM_LIBRARY=yes
```

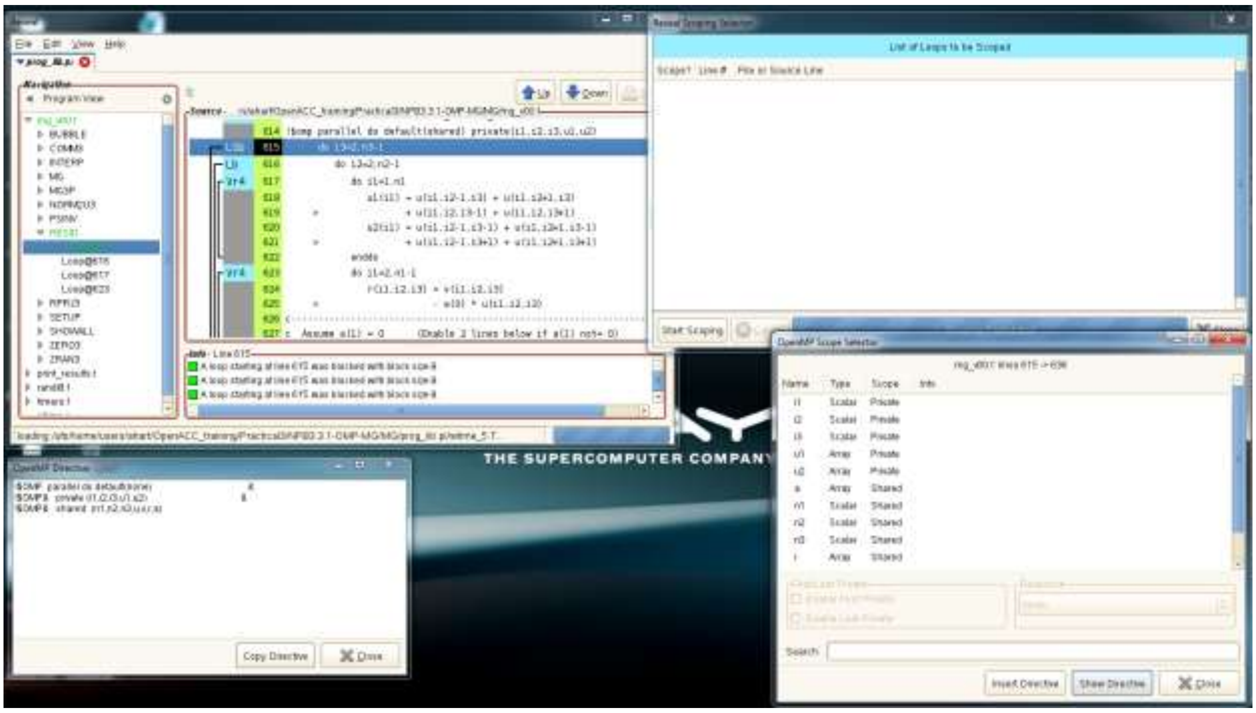
Scoping requires use of the program library. This is a powerful feature which also allows, for instance, much-improved interprocedural optimisation. This requires use of the compiler flags: `-hwp -hpl=<dir>`, which are set by the above option.

The scoping information can then be viewed using the Cray Reveal tool, packaged as part of the `perftools` module:

```
module load perftools
reveal MG/prog_lib.pl
```

This should give a display similar to the main panel below. Using the left navigation pane, we select file `mg_v00.f` and the `resid` function to see the loops. A loopmark-annotated version of the listing is displayed in the main panel. Double-clicking the loop statement in this main panel opens a new window showing the loops we have selected for scoping. Click "Start scoping" to do this. Most variables should be scoped correctly; if not, you can override the compiler's decision. A few variables may not be scoped. Click on these to input the correct treatment. Finally, you can click "Show directive" to see the OpenMP directive for this loop, for cutting and pasting into your code. Note that this directive will not compile if some variables remain unscoped. Alternatively, you can click "Insert directive". (At this point, this is only inserted on the screen within Reveal. You will be given the option later to save this modified version of the code.)

Back in the navigation panel, successfully-scoped loops (all variables scoped) are shown in green in the navigation panel; those where the compiler is not sure are shown in red.



```
614 !$omp parallel do default(shared) private(i1,i2,i3,u1,u2)
615       do i3=2,n3-1
616          do i2=2,n2-1
617             do i1=1,n1
618                u1(i1) = u(i1,i2-1,i3) + u(i1,i2+1,i3)
619      >                + u(i1,i2,i3-1) + u(i1,i2,i3+1)
620                u2(i1) = u(i1,i2-1,i3-1) + u(i1,i2+1,i3-1)
621      >                + u(i1,i2-1,i3+1) + u(i1,i2+1,i3+1)
622             enddo
623             do i1=2,n1-1
624                r(i1,i2,i3) = v(i1,i2,i3)
625      >                     - a(0) * u(i1,i2,i3)
626 c--------------------------------------------------------------------
627 c  Assume a(1) = 0      (Enable 2 lines below if a(1) not= 0)
```

Info - Line 615
🟩 A loop starting at line 615 was blocked with block size 8
🟩 A loop starting at line 615 was blocked with block size 8
🟩 A loop starting at line 615 was blocked with block size 8

```
$OMP  parallel do default(none)                     &
$OMP&   private (i1,i2,i3,u1,u2)                    &
$OMP&   shared  (n1,n2,n3,u,a,r,a)
```

mg_v00.f: lines 615 -> 636

Name	Type	Scope	Info
i1	Scalar	Private	
i2	Scalar	Private	
i3	Scalar	Private	
u1	Array	Private	
u2	Array	Private	
a	Array	Shared	
n1	Scalar	Shared	
n2	Scalar	Shared	
n3	Scalar	Shared	
r	Array	Shared	

4. Step 2. Beginning to add OpenACC directives

You should begin the process by adding two useful features to the code, if they are not already there. The first is a timing routine around the main computational structure. For the MG code, this is the iteration loop in the main program. The MG code already includes such a timer (labeled T_BENCH in the source). Having such a timer is a very non-invasive way of measuring the overall performance of the code. The second is some sort of checksum or correctness check. We need to verify the accelerated version of the code is correct, and it is better to do this at each stage. Again, the MG code already includes such a feature, in the form of a residual which it also checks internally.

4.1. Creating the first OpenACC kernel

The most time-consuming function is `resid`, which contains one loopnest which we will turn into a kernel to execute on the GPU using OpenACC directives. For Fortran, we simply add the directive:

```
!$acc parallel loop private(u1,u2)
```

and its equivalent end directive.

In many cases, we need not specify any data movements explicitly, as the compiler can determine this automatically. Here we discuss an issue with the MG code that prevents this for Fortran.

4.1.1. Fortran standard violations in MG code

In Fortran automatically-determined data movement usually works very well. The MG code has a particular issue, however. One of the calls to this routine takes the form:

```
call resid(u(ir(k)),r(ir(k)),r(ir(k)),m1(k),m2(k),m3(k),a,k)
```

which violates the Fortran standard: the same array is passed twice, once as `INTENT(in)` and once as `INTENT(out)`. Whilst the code fortuitously works on the CPU, it fails with OpenACC with automatically-determined data movements. The real solution is to fix the code but, in the spirit of making as few changes as possible, we can overcome the problem by making the data movements explicit:

```
!$acc parallel loop vector_length(NTHREADS)
!$acc& private(u1,u2) copyin(u,v,a) copyout(r)
```

Note that no array shaping is required in Fortran, although it can be used to move contiguous array slices instead of the full object. The slicing syntax for OpenACC with Fortran is the same as the Fortran standard.

We have also added a clause that controls (in the language of CUDA) the number of threads per block, to be fixed at compile time. For the moment, we will set it to 128 (via compiler flag `-D`), but we can later vary it as a tuning exercise. The Fortran MG code is fixed-form source; for free-form source, we need to end continued lines with `&` (as for OpenMP). Failure to do this can lead to the additional lines being silently ignored, which is a frequent source of error. You can usually diagnose this by looking at the loopmark report for the routine (described below).

4.2. Compiling and running the first kernel

These directives have been implemented in `VERSION=03`. To enable OpenACC support (or rather to stop the Makefile disabling support), make sure that module `craype-accel-nvidia35` is loaded (and `perftools` unloaded) and compile with:

```
make clean
make MG VERSION=03 ACC=yes
```

The code should then be run as before. Doing this, we get:

```
Mop/s total     =                    1541.42     ! Fortran
```

Despite (we hope) the OpenACC version of the kernel being faster, the time spent on associated data transfers to and from the GPU lead to an overall drop in performance.

4.3. Getting information about the OpenACC activity

Information on OpenACC activity is provided at compile time via loopmark, and at runtime through real-time commentary, the NVIDIA Compute Profiler and from CrayPAT[6]. Note that all the runtime methods have an overhead and should not be used when measuring performance.

4.3.1. Loopmark

Evidence that an OpenACC kernel really has been created can be found by looking at the loopmark file (with extension `.lst`). Accelerated regions are marked with letters `G` and/or `g`.

It is clear that a single kernel was created in each case, with the iterations of the outer `i3` loop spread over the threadblocks. The iterations of the inner `i1` loop are divided over the threads in a block, and a blocksize of 128 threads was chosen. The `i2` loop is executed redundantly, meaning that each thread will execute all iterations of this loop for its particular values of `i3` and `i1`. For those familiar with CUDA programming, the equivalent scheduling of the loops would be (assuming tripcounts of each loop are suitably small to avoid unnecessary detail):

```
i3  = blockIdx.x + 1; i1 = threadIdx.x; for (i2=1; i2<n2-1; i2++)
```

Two tasks are carried out by the kernel, populating the temporary arrays `u1`, `u2` and then (with some masking) using these to populate array `r`. Threads in the threadblock are synchronised between these tasks, which for this loop scheduling ensures correctness.

4.3.2. Runtime commentary

Setting environment variable `CRAY_ACC_DEBUG` to values `1`, `2` or `3` in the PBS script when running the code enables a runtime commentary on the OpenACC data movements and kernel execution. It does not need to be set at compile time. Not only does this provide evidence that the kernels really were executed on the GPU, it also provides very useful information on when and where data was transferred to and from the GPU. Setting it will, however, impact performance. The commentary is written to STDERR (collected in the log file) and the larger the value, the more detailed the information displayed.

If you edit the `submit.pbs` file in the previous run directory and uncomment the `CRAY_ACC_DEBUG` line, you can rerun the job (over-writing the previous log file) using:

```
qsub submit.pbs
```

After an initial header, for each instance of the `resid` kernel we see output similar to the following:

```
ACC: Start transfer 6 items from mg_v03.f:615
ACC:       allocate, copy to acc 'a' (32 bytes)
ACC:       allocate 'r' (137388096 bytes)
ACC:       allocate, copy to acc 'u' (137388096 bytes)
ACC:       allocate, copy to acc 'v' (137388096 bytes)
ACC:       allocate <internal> (530432 bytes)
ACC:       allocate <internal> (530432 bytes)
ACC: End transfer (to acc 274776224 bytes, to host 0 bytes)
ACC: Execute kernel resid_$ck_L615_1 blocks:256 threads:128 async(auto) from mg_v03.f:615
ACC: Wait async(auto) from mg_v03.f:639
ACC: Start transfer 6 items from mg_v03.f:639
ACC:       free 'a' (32 bytes)
ACC:       copy to host, free 'r' (137388096 bytes)
ACC:       free 'u' (137388096 bytes)
ACC:       free 'v' (137388096 bytes)
ACC:       free <internal> (0 bytes)
ACC:       free <internal> (0 bytes)
```

[6] If there is no evidence of OpenACC activity, check that module `craype-accel-nvidia35` is loaded when you compile.

```
ACC: End transfer (to acc 0 bytes, to host 137388096 bytes)
```

The large data transfers either side of the kernel are very obvious. In later versions of the CCE, runtime API calls are available to limit the regions in the code where commentary is provided (or to allow the user to select which MPI ranks write commentary).

4.3.3. NVIDIA compute profiler

The CCE interprets OpenACC directives to create GPU kernels written in PTX (a low-level, assembler-like machine language used by NVIDIA GPUs)[7]. As such, we can use tools like the NVIDIA Compute Profiler to gain information about the code performance. To activate this, run the code with environment variable `COMPUTE_PROFILE=1` set in the PBS script (no special compile options are needed). The output is written to a file (by default, `cuda_profile_0.log`) and can provide information on data transfer sizes and times, kernel execution times, occupancy etc. More detailed feedback on, for instance, GPU cache misses can be enabled using a configuration file; see the NVIDIA documentation for more information[8].

The information is provided on a per-kernel basis, with no aggregation or summary. Running this example we get a header, followed by information for each instance of the `resid` kernel:

```
method=[ memcpyHtoD ] gputime=[ 1.088 ] cputime=[ 42.000 ]
method=[ memcpyHtoD ] gputime=[ 52236.543 ] cputime=[ 52513.000 ]
method=[ memcpyHtoD ] gputime=[ 52153.281 ] cputime=[ 52402.000 ]
method=[ resid_$ck_L615_1 ] gputime=[ 15063.424 ] cputime=[ 21.000 ] occupancy=[ 0.333 ]
method=[ memcpyDtoH ] gputime=[ 281508.594 ] cputime=[ 283700.000 ]
```

As the Compute Profiler interprets PTX, it can be used when OpenACC is mixed with CUDA (see below).

4.3.4. CrayPAT profiling

CrayPAT tracing offers a more powerful profiling facility for OpenACC codes. Compilation, instrumenting and running and reporting proceeds exactly as in Sec. 3.1, using `VERSION=03` with `ACC=yes`. First we instrument with command

```
pat_build -w mg.B.x
```

to just show OpenACC kernels. Generating the report with `pat_report -T` (to show all entries) yields two tables. The first aggregates all calls to the `resid` routine, reporting CPU times:

```
Table 1:  Profile by Function Group and Function

 Time%  |     Time   | Imb.  |  Imb.  | Calls  |Group
        |            | Time  | Time%  |        | Function

 100.0% | 16.409900 |    -- |     -- | 1252.0 |Total
|------------------------------------------------------------------------------
| 100.0% | 16.409731 |    -- |     -- |  851.0 |USER
||-----------------------------------------------------------------------------
||  51.2% |  8.403343 |    -- |     -- |    1.0 |mg_
||  34.3% |  5.622111 |    -- |     -- |  170.0 |resid_.ACC_COPY@li.615
||  11.8% |  1.936478 |    -- |     -- |  170.0 |resid_.ACC_COPY@li.639
||   2.7% |  0.440894 |    -- |     -- |  170.0 |resid_.ACC_SYNC_WAIT@li.639
||   0.0% |  0.005727 |    -- |     -- |  170.0 |resid_.ACC_KERNEL@li.615
||   0.0% |  0.001178 |    -- |     -- |  170.0 |resid_.ACC_REGION@li.615
||=============================================================================
|   0.0% |  0.000170 |    -- |     -- |  401.0 |ETC
||-----------------------------------------------------------------------------
```

[7] The CCE does not generate intermediate CUDA, nor is NVCC part of the Cray OpenACC toolchain. It is difficult to preserve debugging symbols (used by, for instance, Allinea DDT) during intermediate language-to-language translations.
[8] Available here.

The two ACC_COPY lines report data movement at either end of the OpenACC parallel region. As the kernel is launched asynchronously, the ACC_KERNEL line is essentially zero and the GPU execution time is measured by the ACC_SYNC_WAIT barrier. The negligible ACC_REGION time here measures internal operations such as setting up pointers for data transfers.

The second table recognises that `resid` is called from two points in the code and performs a similar breakdown for each region. Here is the first part, describing calls to `resid` from subroutine `mg3p`:

```
Table 2:  Time and Bytes Transferred for Accelerator Regions

  Host   | Host   |  Acc   | Acc Copy   | Acc Copy   | Events   |Calltree
 Time%   | Time   | Time   |         In |        Out |          |
         |        |        | (MBytes)   | (MBytes)   |          |

 100.0% | 8.007 | 7.962 |      12341 |       6171 |      850 |Total
|-----------------------------------------------------------------------------------------
| 100.0% | 8.007 | 7.962 |      12341 |       6171 |      850 |mg_
||----------------------------------------------------------------------------------------
||  50.0% | 4.005 | 3.969 |       6314 |       3157 |      735 |mg3p_
3|        |       |       |            |            |          | resid_
4|        |       |       |            |            |          |  resid_.ACC_REGION@li.615
||||------------------------------------------------------------------------------------
5||||  36.2% | 2.898 | 2.877 |       6314 |         -- |      147 |resid_.ACC_COPY@li.615
5||||  10.8% | 0.867 | 0.860 |         -- |       3157 |      147 |resid_.ACC_COPY@li.639
5||||   2.9% | 0.235 |    -- |         -- |         -- |      147 |resid_.ACC_SYNC_WAIT@li.639
5||||   0.1% | 0.004 | 0.232 |         -- |         -- |      147 |resid_.ACC_KERNEL@li.615
5||||   0.0% | 0.001 |    -- |         -- |         -- |      147 |resid_.ACC_REGION@li.615(exclusive)
```

Times are also given for both CPU and GPU. The ACC_KERNEL is asynchronous as the CPU time is very small. It is then clear that the ACC_SYNC_WAIT on the CPU includes the time spent by the GPU on the kernel. The exclusive ACC_REGION time here is the same as in the previous table.

To see how the OpenACC events feature in the full profile, reinstrument with:

```
pat_build -u mg.B.x
```

which gives a report containing:

```
Table 1:  Profile by Function Group and Function

 Time%   |      Time   | Imb.   |  Imb.   |    Calls   |Group
         |             | Time   | Time%   |            | Function

 100.0% | 16.452925 |    -- |      -- | 265303.0 |Total
|----------------------------------------------------------------------------
| 100.0% | 16.452760 |    -- |      -- | 264902.0 |USER
||---------------------------------------------------------------------------
||  34.2% |  5.621172 |    -- |      -- |    170.0 |resid_.ACC_COPY@li.615
||  19.4% |  3.199216 |    -- |      -- |    168.0 |psinv_
||  11.8% |  1.940111 |    -- |      -- |    170.0 |resid_.ACC_COPY@li.639
||  10.7% |  1.764268 |    -- |      -- | 131072.0 |vranlc_
||   7.4% |  1.217534 |    -- |      -- |    147.0 |rprj3_
||   6.3% |  1.033920 |    -- |      -- |    147.0 |interp_
||   4.3% |  0.709337 |    -- |      -- |    151.0 |zero3_
||   2.7% |  0.441237 |    -- |      -- |    170.0 |resid_.ACC_SYNC_WAIT@li.639
||   1.5% |  0.240856 |    -- |      -- |      2.0 |zran3_
||   1.0% |  0.170554 |    -- |      -- |    487.0 |comm3_
|============================================================================
```

Clearly the `resid` kernel is now no longer dominant in the profile, but the associated data copies to and from the GPU are very expensive. To see all the smaller entries, use `pat_report -T`.

4.3.5. Profiling with GPU performance counters

CrayPAT supports a wide range of accelerator performance counters. To learn more about this, type (when the `perftools` module is loaded):

```
man accpc
more $CRAYPAT_ROOT/share/CounterGroups.nvidia_k20x
```

To use these, the code is compiled and instrumented (for tracing) as before. An environment variable is then set at runtime (in the PBS script) to indicate the collection of one or more performance counters e.g. `PAT_RT_ACCPC= ipc_inst_rep_ovr`. This makes `pat_report` generate a third table:

```
Table 3:  ACC Performance Counter Data

   Acc    | inst_executed   | inst_issued1   | inst_issued2   | thread_inst_executed   |  ipc   |  Acc   |Calltree
 Time%    |                 |                |                |                        |        | Util   |

 100.0% |      273341540 |     245524426 |      54206381 |             8261569560 | 0.003 | 45.5% |Total
|-----------------------------------------------------------------------------------------------------------------------
| 100.0% |      273341540 |     245524426 |      54206381 |             8261569560 | 0.003 | 45.5% |mg_
||----------------------------------------------------------------------------------------------------------------------
||  50.2% |      129718820 |     117634433 |      25615019 |             3940872320 | 0.003 | 22.9% |resid_
3|        |                |               |               |                        |       |       | resid_.ACC_REGION@li.615
||||--------------------------------------------------------------------------------------------------------------------
4|||  34.1% |              0 |             0 |             0 |                      0 |     0 | 15.5% |resid_.ACC_COPY@li.615
4|||  13.5% |              0 |             0 |             0 |                      0 |     0 |  6.1% |resid_.ACC_COPY@li.639
4|||   2.6% |      129718820 |     117634433 |      25615019 |             3940872320 | 0.061 |  1.2% |resid_.ACC_KERNEL@li.615
||||====================================================================================================================
||  49.8% |      143622720 |     127889993 |      28591362 |             4320697240 | 0.004 | 22.7% |mg3p_
3|        |                |               |               |                        |       |       | resid_
4|        |                |               |               |                        |       |       |  resid_.ACC_REGION@li.615
|||||-------------------------------------------------------------------------------------------------------------------
5||||  36.1% |              0 |             0 |             0 |                      0 |     0 | 16.4% |resid_.ACC_COPY@li.615
5||||  10.8% |              0 |             0 |             0 |                      0 |     0 |  4.9% |resid_.ACC_COPY@li.639
5||||   2.9% |      143622720 |     127889993 |      28591362 |             4320697240 | 0.060 |  1.3% |resid_.ACC_KERNEL@li.615
|======================================================================================================================
```

5. Step 2: More kernels

Having created one OpenACC kernel for `resid`, it is relatively straightforward to port all four of the kernels dominating the profile. This has been done in `VERSION=04`. Compiling and running this, we find for Fortran:

```
Mop/s total      =                    1274.48      ! Fortran
```

With even more data transfers, the code now executes even more slowly, but does give the right answer.

6. Step 3: Reducing data movements

Having ported all of the major kernels, we would like to introduce a data region to avoid data being needlessly transferred back from the GPU and then over again between kernels. Reducung data transfers is usually the single biggest optimisation that can be made. The obvious place for this data region is around the main iteration loop.

There are two steps to this. Firstly, we need to make sure that all routines that operate on the data are ported to the GPU, not just the ones that seem most significant in the profile. Otherwise, data will have to be shipped back to the CPU simply to execute an "insignificant" operation, which can be very damaging for performance.

We therefore need to add OpenACC directives to `comm3`, `zero3`, `norm2u3`. This has been done in `VERSION=05`. Once more, the code can be executed to check for correctness. The performance is now significantly worse:

```
Mop/s total      =                     886.02      ! Fortran
```

6.1. Introducing a data region

We now introduce the data region. We can arrange for the main data arrays to be initialised on the GPU, and the final value of the arrays is not needed. So we specify the main data arrays `u`,`v`,`r` using the `create` clause. The parameter arrays a, c are initialised on the CPU (using sequential code) and so we copy the values to the GPU using the copyin clause.

To ensure the existing OpenACC kernels use the GPU-resident versions of the array, we change the copy, copyin and copyout clauses (collectively called copy* here) to present_or_copy*, or pcopy* for short. In cases where the kernels are known to only be called within a data region, it is sometimes better to replace the copy* clauses with present; any data scoping errors (where kernels are called outside the data region) will lead to a runtime code failure at a specific line, rather than just a wrong answer at the end of the run (which would have to be diagnosed by trawling the data movement log from CRAY_ACC_DEBUG).

The final change needs to be made in the zran3 initialisation routine. A random number generator and a bubblesort are used to populate an array jg that is then used to initialise the data array z. Porting these to the GPU, even with OpenACC, is tricky, so we opt to execute this code on the host and then copy jg to the GPU at the end to populate z. The one complication is that array z is has previously been used as a temporary array during the creation of jg. This is not necessarily a problem; the versions of z in CPU and GPU memories are distinct, and we can use the CPU version as the temporary array. We therefore need to make sure that it is this version that is zeroed at the start of zran3, so we replace the call to zero3 (which will execute on the GPU) by an explicit loopnest to execute on the CPU. Conversely, when the array z is zeroed at the end of the routine, this should refer to the GPU version. So here we replace the explicit loopnest by a call to zero3.

Whilst these changes are specific to the MG code, they are, however, typical of the sorts of code issues encountered when accelerating a real application with OpenACC directives, so we make no apology for discussing this.

These changes have been made in VERSION=06. Compiling and running this, we have a correct result and the performance is:

```
Mop/s total     =                 23913.21     ! Fortran
```

Finally we have a version that has better performance than the single CPU core with a speedup of nearly 8x. If we collect the CRAY_ACC_DEBUG output in a file (e.g. stderr), grep-ing for “copy” shows that we have eliminated all movement of the main data arrays. Assuming we are in a bash shell:

```
aprun <options> env CRAY_ACC_DEBUG=2 bin/mg.B.x 2> stderr
grep "copy" stderr | sort | uniq
```

gives (for the Fortran version):

```
ACC:       allocate, copy to acc 'a' (32 bytes)
ACC:       allocate, copy to acc 'c' (32 bytes)
ACC:       allocate, copy to acc 'jg' (320 bytes)
ACC:       allocate reusable, copy to acc <internal> (16 bytes)
ACC:       allocate reusable, copy to acc <internal> (4 bytes)
ACC:       copy to host, done reusable <internal> (16 bytes)
ACC:       reusable acquired, copy to acc <internal> (16 bytes)
```

7. Step 4: Performance tuning

Once data movements have been minimised, attention should be turned to kernel performance tuning.

The simplest performance tuning is to vary the number of threads per block. This can be varied globally by recompiling with option NTHREADS=<VALUE> (later versions of CCE may support a compiler option). Supported values are powers of two between 64 and 1024 (i.e. 64, 128 (default), 256, 512 and 1024). The performance gain can be maximised by profiling the code for each number of threads to determine the optimal number on a kernel-by-kernel basis and then to set vector_length individually for each significant accelerator region. For the MG code, this gain is, however, modest. Looking at the four most significant kernels in VERSION=06, we gain only 2% in overall performance relative to the default, 128 threads.

Optimising kernel performance normally reduces to adjusting the loop structure to better fit to the GPU architecture. Examining loopmark is crucial in this. The fastest-moving loop should be scheduled across the vector level of parallelism (to aid coalesced memory accesses). Collapsing loops (with the collapse clause) often improves performance. This requires the loops to be collapsed to be perfectly nested which can give other performance gains). Removing the private temporary arrays (by manual inlining) in the MG kernels can help with this.

To control scheduling of individual loops, loop directives can be nested within the parallel region, with the gang, worker and vector clauses used as appropriate.

An example of a moderately tuned resid kernel is given in VERSION=07, which improves performance:

```
Mop/s total     =                   23999.08     ! Fortran
```

The gain is, however, so small that it is questionable whether this tuning of the source code was worth it.

7.1. Calling external CUDA kernels

A more extreme form of optimisation is to call kernels written in the lower-level NVIDIA CUDA language. OpenACC is compatible with CUDA: we can call CUDA kernels (via a C wrapper function) and pass pointers to data that is already resident in the GPU memory to avoid data copies. This allows the user to call, for instance, hand-optimised CUDA code for critical kernels or accelerated libraries.

To do this, we use the host_data directive to instruct the compiler to expose the location of the data in GPU global memory.

We illustrate this in VERSION=08. If we compile with option CUDA=yes, a CUDA version of resid is included. The CUDA (and associated C wrapper) is compiled with nvcc, and the code is linked with CCE:

```
make clean
make MG VERSION=08 ACC=yes CUDA=yes
```

The CUDA kernel is very naïve, and does not improve the performance (which is a cautionary lesson):

```
Mop/s total     =                   22351.51     ! Fortran
```

The NVIDIA Compute Profiler can be used to verify that the CUDA kernel is being called.

8. Conclusions

We started with a Fortran code with 1445 lines (of which 267 were blank). The code was fully ported to the accelerator using 16 directives.

Using the GPU has given a 15x speedup of the MG code compared to a single core of an AMD Interlagos CPU. To justify the OpenACC port, we should compare with the performance of the original code using OpenMP across the full node:

```
make clean
make MG OMP=yes
```

You can then submit the code using the same submission script, adding an additional argument:

```
bash submit.bash bin/mg.B.x <N>
```

where <N> is the number of threads, up to 16 on a Cray XK7 or 32 on a Cray XE6 (e.g. rosa). Doing this, we get:

```
Mop/s total     =                    9162.37     ! Fortran, Cray XK7, 16 threads
Mop/s total     =                   16313.24     ! Fortran, Cray XE6, 32 threads
```

The OpenACC version is therefore 2.6x faster than a single AMD Interlagos CPU, and 1.5x faster than a dual-socket Cray XE6 node. This is very encouraging, with many optimisation avenues still unexplored. The CrayPAT loop profiling statistics suggest that we will find it hard to realise the full power of the GPU: even for large local problem sizes, many of the stencil kernels are called with very low trip counts. To really get good GPU performance may require more radical application restructuring; such changes are, however, markedly easier with OpenACC than in a lower-level programming model like CUDA.

In general, comparing with floating point performance or memory bandwidth, the NVIDIA Kepler K20x GPU is around a factor of 5-10x faster than a single 16-core AMD Interlagos CPU, or 2.5x faster than a Cray XE6 node. This is what we see for many applications, running either with CUDA or OpenACC on the GPU, either in serial or parallel.

For some codes, the OpenACC `async` clause can offer additional performance gains through the overlap of GPU computation and data transfers with CPU computation and network transfers. For the MG code, however, there is almost no scope for using this; each computational task depends on the results of the one before, so we cannot overlap kernels.

The issue of low trip counts can be addressed in the code through use of the `collapse` clause or, for triply-nested loops, using the `worker` clause for the middle loop. More radically, the multigrid algorithm is used to speed convergence. If the coarsest grids (i.e. those with lowest loop tripcounts) are detrimental to performance, should they be included in a GPU version?

MG is a finite difference, stencil code and these codes can sometimes benefit from use of the `cache` clause, which requires loop blocking or use of the `tile` clause planned for OpenACC v2.0.