Porting real applications

Alistair Hart
Cray Exascale Research Initiative Europe

Adding OpenACC to a Larger Code

- **Adding OpenACC to a real code is not trivial work…**
  - Are parts of the program suitable for an accelerator?
  - Where do we start?
  - What do we do next?

- **We'll go through the exercise for an example code**
  - Running on a Cray XK7 (AMD Interlagos and Nvidia Kepler K20x)
  - Using Cray compiler and Cray performance analysis tools

The Code

- **NAS Parallel Benchmarks MG (MultiGrid) code**
  - Shorter than typical application
    - but structure of code is very similar
  - This example concentrates on the serial version
    - We also have parallel versions ported to OpenACC
    - The serial versions have OpenMP directives, but we do not use them during this exercise
  - Downloading it:
    - Fortran version: http://www.nas.nasa.gov/publications/npb.html.
      - 1445 lines, of which 267 blank
    - C version: http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp/download/download-benchmarks.html.
      - 1292 lines, of which 206 blank

Building and Running MG

- **Build:**
  - make MG [CLASS=<CLASS>] [<OPTIONS>]
    - CLASS is the problem size. "B" is the default.
  - Top-level Makefile passes options to MG/Makefile
    - this uses config/make.def for compiler-specific options

- **Run:**
  - Three important lines of output
    - Fortran
      - `L2 Norm is  0.1800564401355E-05`
      - `Mop/s total     =                    1623.04`
      - `Verification    =                 SUCCESSFUL`
    - C output same, but baseline performance differs
      - `Mop/s total     =                    1320.26`
  - Always check:
    - L2 Norm should not be a NaN
    - Verification should be successful

Where Do We Start?

- **Profile MG on the CPU**

```
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_
|==========================================================
```

- **Four routines dominate the runtime**
  - More than half the time is spent in **resid**
  - There are other routines executing for less than 1% of the total time
    - These might be important for the OpenACC port

Understand Flow of the Application

```
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_
|=====================================
```

- **mg calls:**
  - mg3p (which then calls resid, psinv, rprj3, interp)
  - resid also called directly from mg

Get Work Estimates for Loops

```
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
```

- **Loop-level profiling is more useful now**
  - Which loopnests (rather than just routines) took most time?
  - How may iterations did this loopnest have?
- **Here are the lines relating to resid**
  - Loops starting at 636 and 642 are nested inside loops at line 634, 635
    - See how the Loop Hit numbers multiply up
    - See how inclusive times for 636 and 642 add to give that for 635
    - Inclusive times for 634, 635 same: perfectly nested loops

Add First OpenACC Kernel

- **Clearly we should start with resid**

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

  - C:
    - `#pragma acc parallel loop vector_length(NTHREADS) \`
      `private(u1,u2) copyin(u[0:n1*n2*n3],v[0:n1*n2*n3],a[0:4]) \`
      `copyout(r[0:n1*n2*n3])`

    - Data movement sizes explicit to avoid "unshaped pointer" errors
      - Because we are dynamically allocating memory

Resulting MG Performance?

- Running with and without OpenACC kernel:

	Original (Mop/s)	1 kernel (Mop/s)
Fortran	1623.04	1541.42
C	1320.26	1409.48

- So the code is actually slower... **Why?**

Enable Cray Runtime Commentary

- export CRAY_ACC_DEBUG=2
- for every call to resid:

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

- **Certainly a lot of data was moved**
  - Commentary tells us which arrays, at which line and how much data

Or Use Nvidia Compute Profiler

- **export COMPUTE_PROFILE=1**
  - Analyses PTX (from OpenACC or from CUDA)
  - Very useful if mixing OpenACC with CUDA code

```
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 ]
```

- **Data transfers obvious, taking most time**

Or Use CrayPAT for a Profile by Function

```
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
||----------------------------------------------------------------------
```

- **Provides aggregated report of data movements**
  - names, sizes and frequencies of original arrays lost
- **Shows asynchronous kernel launches**
  - Notice ACC_KERNEL almost zero
  - SYNC_WAIT shows the compute time
  - could recompile with -hacc_model=auto_async_none

…And CrayPAT Accelerator Statistics

```
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)
```

- **Host and accelerator times given separately**
  - ACC_KERNEL
    - Acc Time is the compute time
    - Host Time is the time for the asynchronous launch
      - The Host "catches up" at the SYNC_WAIT

… And CrayPAT Summarized Trace

- pat_build -u mg.B.x

```
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_
|==========================================================================
```

- **resid kernel no longer dominates the profile**
  - actual compute time is shown in SYNC_WAIT (Host) Time
  - Its data copies are significant, however

More OpenACC Kernels

- Running with 4 accelerated kernels:

	Original (Mop/s)	1 kernel (Mop/s)	4 kernels (Mop/s)
Fortran	1623.04	1541.42	1274.48
C	1320.26	1409.48	991.42

- Even slower, and C particularly bad. **Why?**

Profile the Code Again

- **Notice that spending most time in interp**
- **Next look at compiler listing:**

```
727.                #pragma acc parallel loop private(z1,z2,z3)  \
728.                  copy(u[0:n1*n2*n3])                        \
729.                  copyin(z[0:mm1*mm2*mm3])
730.  gG------<>         for (i3 = 0; i3 < mm3-1; i3++) {
731.  1--------<                for (i2 = 0; i2 < mm2-1; i2++) {
732.  1 g------<                 for (i1 = 0; i1 < mm1; i1++) {
733.  1 g                          i123 = i1 + mm1*i2 + mm12*i3;
734.  1 g                            z1[i1] = z[i123+mm1] + z[i123];
735.  1 g                            z2[i1] = z[i123+mm12] + z[i123];
736.  1 g                            z3[i1] = z[i123+mm1+mm12] + z[i123+mm12] + z1[i1];
737.  1 g------>                 }
738.  1 r4-----<                 for (i1 = 0; i1 < mm1-1; i1++) {
739.  1 r4                         i123 = i1 + mm1*i2 + mm12*i3;
740.  1 r4                         j123 = 2*i1 + n1*(2*i2 + n2 * 2*i3);
741.  1 r4                           u[j123] += z[i123];
742.  1 r4                           u[j123+1] += 0.5*(z[i123+1]+z[i123]);
743.  1 r4----->                 }
```

Loop Accelerated

Loop Partitioned

Loop Not Partitioned

- **Loop at line 738 not partitioned**
  - Executed redundantly: every thread does every loop iteration

More OpenACC Kernels

- **Insert directive above unpartitioned loop to help compiler:**
  - #pragma acc loop independent
  - (if this didn't work, would use "#pragma acc loop vector" instead

	Original (Mop/s)	1 kernel (Mop/s)	4 kernels (Mop/s)
Fortran	1623.04	1541.42	1274.48
C	1320.26	1409.48	1271.12

- **Fortran and C performance now identical**

Next Steps – Reduce Data Movement

- **Need to introduce data regions higher up calltree**
- **For this, need all callee routines to be accelerated with OpenACC directives**
- **Use a profiler to map out the calltree to get list of routines**
- **First need to port some insignificant routines**
  - norm2u3, zero3, comm3

Results for Accelerating up the Calltree

- **Accelerated loops in norm2u3, zero3, comm3**

	Original (Mop/s)	1 kernel (Mop/s)	4 kernels (Mop/s)	Calltree Routines (Mop/s)
Fortran	1623.04	1541.42	1274.48	886.02
C	1320.26	1409.48	1271.12	873.01

- **Slower because even more data movement**
  - C still slightly down; poor scheduling needs acc loop independent

	Original (Mop/s)	1 kernel (Mop/s)	4 kernels (Mop/s)	Calltree Routines (Mop/s)
Fortran	1623.04	1541.42	1274.48	886.02
C	1320.26	1409.48	1271.12	885.27

Add Data Region

- **Now we put a data region in the main program**
  - Arrays u,v,r are declared create
    - We'll never use the host version of these
  - Arrays a,c are declared copyin
    - They're initialized on the host

- **Then in all the subprograms, we change clauses**
  - Replace copy* and create by present
    - Could replace by present_or_*
      - If we know they should always be present, better to state this
      - Then mistakes become runtime errors rather than just wrong answers
        - We'd have to diagnose these by trawling the runtime commentary

Results with Data Region in Main

- At last, we are running faster (and correctly)!

	Original (Mop/s)	1 kernel (Mop/s)	4 kernels (Mop/s)	Calltree Routines (Mop/s)	Data Region (Mop/s)
Fortran	1623.04	1541.42	1274.48	886.02	23913.21
C	1320.26	1409.48	1271.12	885.27	23558.03

View Compiler Commentary Again

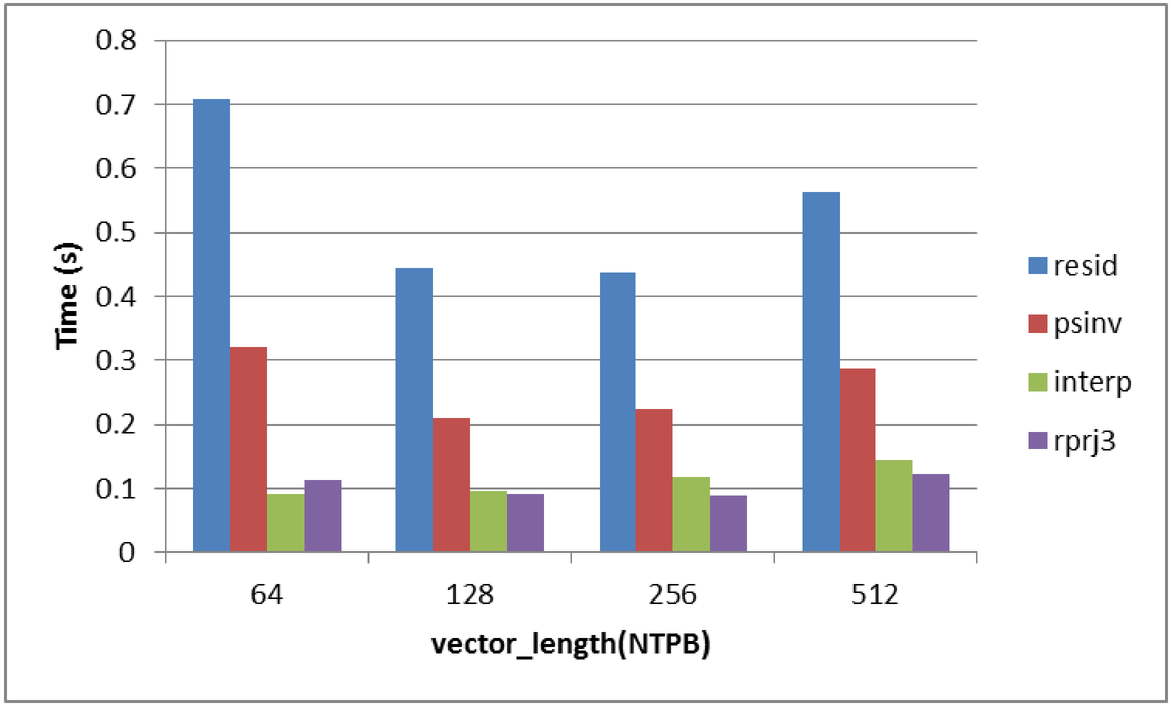
- **All array data transfers eliminated**
  - Run with CRAY_ACC_DEBUG=2 and catch STDERR in a file
  - grep "copy" <file> | sort | uniq

```
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)
```

  - Arrays a, c, jg copied only at initialization
  - Some internal transfers unavoidable

Performance Tuning Tips

- **Check the .lst loopmark file**
  - Are any kernels obviously badly scheduled?
    - No, we already checked that
- **Try varying vector_length from the default of 128**
  - Different values may suit different kernels
  - Effect small here: up to 5% per kernel, but only 2% overall:



Performance Tuning Tips

- **Loop scheduling**
  - Collapse loops within loopnests
    - OpenACC schedules according to the loops in the nest
    - Collapsing loops can increase the tripcount
      - e.g. to allow more threads in a block
  - Using the worker clause may help for imperfectly nested loops

  - Block loops in a loopnest
    - This can improve cache usage (as on the CPU)
    - CCE-specific directives can do this, or try it manually

- **More extreme:**
  - Avoid temporary arrays
    - Use private scalars, as these are more likely to go into registers

  - Rewrite the most expensive kernels in CUDA and handtune
    - but remember you are competing against a whole compiler team

Other Performance Tuning Improvements

- **Avoiding temporary arrays in resid:**
  - `Mop/s total      =                          23999.08 !  Fortran`
  - `Mop/s total      =                          23640.24 // C`

  - benefit v. small; was it really worth hacking the source?
- **Call an external CUDA version of resid**
  - `Mop/s total      =                          22351.51 !  Fortran`
  - `Mop/s total      =                          21289.72 // C`

  - This was a naive kernel
    - (Even so, there may be a lesson in this)
- **Data movement was a far bigger optimisation**
  - than any of our kernel improvements

Conclusions: How far did we get?

- **Significant speedup compared to single core:**
  - Fortran: 1826.19 → 23999.08 = 15x
  - C: 1320.26 → 23640.24 = 18x
- **The real comparison is to a full CPU or node**
  - run the OpenMP version of the code
    - across 16 cores for an XK6 node (single AMD Interlagos)
      - `Mop/s total     =     9162.37 !  Fortran, Cray XK7 CPU, 16 threads`
      - `Mop/s total     =     8638.68 // C,       Cray XK7 CPU, 16 threads`
    - across 32 cores for an XE6 node (dual AMD Interlagos)
      - `Mop/s total     =    15244.09 !  Fortran, Cray XE6, 32 threads`
      - `Mop/s total     =    15120.86 // C,       Cray XE6, 32 threads`
  - maybe MPI version would scale better, but our code here is scalar
- **Final speedup compared to full node:**
  - XK7 CPU node (16 cores): **2.6x**
  - XE6 CPU node (32 cores): **1.5x**
    - This is for Fortran, C looks slightly better

Conclusions: How much further could we get?

- **So we are 2.6x or 1.5x faster, node-for-node**
  - How much faster could we get (speeds and feeds)
    - Flops and mem. b/w around 5-10x faster than single AMD Interlagos
- **So why were we not faster?**
  - MultiGrid application cycles through grid sizes
    - sometimes the grid is really small: 4x4x4
    - CrayPAT loop profiling showed us that
  - Small grid sizes will never schedule well on the GPU
    - consider checking grid size and only using OpenACC for larger ones
    - or do we even need the smaller grids?