Worked example: the scalar Himeno code

Alistair Hart

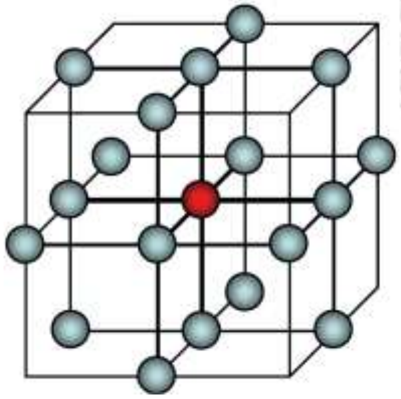
Cray Exascale Research Initiative Europe

Overview

- **This worked example leads you through accelerating a simple application**
  - a simple application is easy to understand
  - but it shows all the steps you would use for a more complicated code

- **In the following practical, you will work through these steps yourself**

The Himeno Benchmark



- **3D Poisson equation solver**
  - Iterative loop evaluating 19-point stencil
  - Memory intensive, memory bandwidth bound
- **Fortran and C implementations available from http://accc.riken.jp/2444.htm**
- **We look at the scalar version for simplicity**
- **Code characteristics**
  - Around 230 lines of Fortran or C
  - Arrays statically allocated
    - problem size fixed at compile time

Why use such a simple code?

- **Understanding a code structure is crucial if we are to *successfully* OpenACC an application**
  - i.e. one that runs faster node-for-node (not just GPU vs. single CPU core)

- **There are two key things to understand about the code:**
  - How is data passed through the calltree?
    - CPUs and accelerators have separate memory spaces
    - The PCIe link between them is relatively slow
    - Unnecessary data transfers will wipe out any performance gains
    - A successful OpenACC port will keep data resident on the accelerator
  - Where are the hotspots?
    - The OpenACC programming model is aimed at loop-based codes
      - Which loopnests dominate the runtime?
      - Are they suitable for a GPU?
        - What are the min/average/max tripcounts?
    - Minimising data movements will probably require eventual acceleration of many more (and possibly all) loopnests, but we have to start somewhere

- **Answering these questions for a large application is hard**
  - There are tools to help (we will discuss some of them later in the course)
  - With a simple code, we can do all of this just by code inspection

Stages to accelerating an application

1. **Understand and characterise the application**
   - Profiling tools, code inspection, speaking to developers if you can
2. **Introduce first OpenACC kernels**
3. **Introduce data regions in subprograms**
   - reduce unnecessary data movements
   - will probably require more OpenACC kernels
4. **Move up the calltree, adding higher-level data regions**
   - ideally, port entire application so data arrays live entirely on the GPU
   - otherwise, minimise traffic between CPU and GPU
   - This will give the single biggest performance gain
5. **Only now think about performance tuning for kernels**
   - First correct any obviously inefficient scheduling on the GPU
     - This will give some good performance improvements
   - Optionally, experiment with OpenACC tuning clauses
     - You may gain some final additional performance from this

- **And remember Amdahl's law...**

Step 1: Himeno program structure

- **Code has two subprograms**
  - init_mt() initialises the data array
    - Called once at the start of the program
  - jacobi() performs iterative stencil updates of the data array
    - The number of updates is an argument to the subroutine and fixed
      - A summed residual is calculated, but not tested for convergence
    - This subroutine is called twice, and each call is timed:
      - Each call is timed internally by the code
      - The first call does a small fixed number of iterations.
        - The time is used to estimate how many iterations could be done in one minute
      - The second call does this number of iterations
        - The time is converted into a performance figure by the code

        - Actually, it is useful when testing to do a fixed number of iterations
        - Then we can use the value of the residual for a correctness check.

  - The next slide shows an edited version of the code
    - These slides discuss the Fortran version; there is also a C code

Step 1: Himeno program structure (contd)

```
PROGRAM himeno
   INCLUDE "himeno_f77.h"

   CALL initmt      ! Initialise local matrices

   cpu0 = gettime() ! Wraps SYSTEM_CLOCK
   CALL jacobi(3,gosa)
   cpu1 = gettime()
   cpu = cpu1 - cpu0

!  nn = INT(ttarget/(cpu/3.0)) ! Fixed runtime
   nn = 1000        ! Hardwired for testing

   cpu0 = gettime()
   CALL jacobi(nn,gosa)
   cpu1 = gettime()
   cpu = cpu1 - cpu0
   xmflops2 = flop*1.0d-6/cpu*nn

   PRINT *,' Loop executed for ',nn,' times'
   PRINT *,' Gosa :',gosa
   PRINT *,' MFLOPS:',xmflops2,'  time(s):',cpu
END PROGRAM himeno
```

- In the next slides we look at the details of jacobi()

Step 1: Structure of the jacobi routine

- Outer loop is executed fixed number of times
  - loop must be sequential !
- Apply stencil to p to create temporary wrk2
  - residual gosa computed
    - details on the next slide
- Pressure array p updated from wrk2
  - this loopnest can be parallelised
- Outer halo of p is fixed

```
SUBROUTINE jacobi(nn,gosa)

  iteration: DO loop = 1, nn

! compute stencil: wrk2, gosa from p
     <described on next slide>

! copy back wrk2 into p
      DO k = 2,kmax-1
         DO j = 2,jmax-1
            DO i = 2,imax-1
               p(i,j,k) = wrk2(i,j,k)
            ENDDO
         ENDDO
      ENDDO

  ENDDO iteration

END SUBROUTINE jacobi
```

Step 1: The Jacobi computational kernel

- The stencil is applied to pressure array p
  - 19-point stencil
- Updated pressure values are saved to temporary array wrk2
- Residual value gosa
  - computed here
- This loopnest dominates runtime
  - Can be computed in parallel
  - gosa is reduction variable

```
gosa = 0
DO k = 2,kmax-1
 DO j = 2,jmax-1
  DO i = 2,imax-1
   s0=a(i,j,k,1)*p(i+1,j, k ) &
     +a(i,j,k,2)*p(i, j+1,k ) &
     +a(i,j,k,3)*p(i, j, k+1) &
     +b(i,j,k,1)*(p(i+1,j+1,k )-p(i+1,j-1,k )  &
                 -p(i-1,j+1,k )+p(i-1,j-1,k )) &
     +b(i,j,k,2)*(p(i, j+1,k+1)-p(i, j-1,k+1)  &
                 -p(i, j+1,k-1)+p(i, j-1,k-1)) &
     +b(i,j,k,3)*(p(i+1,j, k+1)-p(i-1,j, k+1)  &
                 -p(i+1,j, k-1)+p(i-1,j, k-1)) &
     +c(i,j,k,1)*p(i-1,j, k ) &
     +c(i,j,k,2)*p(i, j-1,k ) &
     +c(i,j,k,3)*p(i, j, k-1) &
     + wrk1(i,j,k)

   ss = (s0*a(i,j,k,4)-p(i,j,k)) * bnd(i,j,k)
   gosa = gosa + ss*ss
   wrk2(i,j,k) = p(i,j,k) + omega*ss
  ENDDO
 ENDDO
ENDDO
```

fwd n.n.

n.n.n.

bwd n.n.

Step 2: a first OpenACC kernel

- Start with most expensive
  - apply parallel loop
  - end parallel loop optional
    - *advice: use it for clarity*
- reduction clause
  - like OpenMP, not optional
- private clause
  - loop variables default private (like OpenMP)
  - scalar variables default private (unlike OpenMP)
  - so clause optional here
    - *advice: use one for clarity*
- copy* data clauses
  - compiler will do automatic analysis
  - explicit clauses will interfere with data directives at next step
    - *advice: only use if compiler over-cautious*

```
gosa1 = 0

!$acc parallel loop reduction(+:gosa1) &
!$acc&  private(i,j,k,so,ss) &
!$acc&  copyin(p,a,b,c,bnd,wrk1) &
!$acc&  copyout(wrk2)
DO k = 2,kmax-1
 DO j = 2,jmax-1
  DO i = 2,imax-1
   s0 = a(i,j,k,1) * p(i+1,j, k ) &
     <etc...>

   ss = (s0*a(i,j,k,4) - p(i,j,k)) * &
                              bnd(i,j,k)
   gosa1 = gosa1 + ss*ss
   wrk2(i,j,k) = p(i,j,k) + omega*ss
  ENDDO
 ENDDO
ENDDO
!$acc end parallel loop
```

Compiler feedback

- **Compiler feedback is extremely important**
  - Did the compiler recognise the accelerator directives?
    - A good sanity check
  - How will the compiler move data?
    - Only use data clauses if the compiler is over-cautious on the copy*
    - Or you want to declare an array to be scratch (create clause)

    - The first main code optimisation is removing unnecessary data movements
  - How will the compiler schedule loop iterations across GPU threads?
    - Did it parallelise the loopnests?
    - Did it schedule the loops sensibly?

    - The other main optimisation is correcting obviously-poor loop scheduling

- **Compiler teams work very hard to make feedback useful**
  - advice: use it, it's free! (i.e. no impact on performance to generate it)
    - CCE: -hlist=a Produces commentary files <stem>.lst
    - PGI: -Minfo Feedback to STDERR

g = partitioned loop

G = accelerator kernel

```
163. 1--------< DO loop = 1,nn
169. 1           gosa1 = 0
171. 1 G-----<> !$acc parallel loop reduction(+:gosa1) private(i,j,k,s0,ss)
172. 1 g------<  DO k = 2,kmax-1
173. 1 g 3----<   DO j = 2,jmax-1
174. 1 g 3 g--<    DO i = 2,imax-1
175. 1 g 3 g         s0 = a(i,j,k,1) * p(i+1,j,k) ...
188. 1 g 3 g-->    ENDDO
189. 1 g 3---->   ENDDO
190. 1 g------>  ENDDO
191. 1          !$acc end parallel loop
208. 1--------> ENDDO
```

Numbers denote serial loops

source line numbers

```
163. 1--------< DO loop = 1,nn
169. 1            gosa1 = 0
171. 1 G-----<> !$acc parallel loop reduction(+:gosa1) private(i,j,k,s0,ss)
172. 1 g------<  DO k = 2,kmax-1
173. 1 g 3----<   DO j = 2,jmax-1
174. 1 g 3 g--<    DO i = 2,imax-1
175. 1 g 3 g         s0 = a(i,j,k,1) * p(i+1,j,k) ...
188. 1 g 3 g-->    ENDDO
189. 1 g 3---->   ENDDO
190. 1 g------>  ENDDO
191. 1          !$acc end parallel loop
208. 1--------> ENDDO

Data movements:
ftn-6418 ftn: ACCEL File = himeno_f77_v02.f, Line = 171
  If not already present: allocate memory and copy whole array "p" to accelerator,
   free at line 191 (acc_copyin).

<identical messages for a,b,c,wrk1,bnd>

ftn-6416 ftn: ACCEL File = himeno_f77_v02.f, Line = 171
  If not already present: allocate memory and copy whole array "wrk2" to accelerator,
   copy back at line 191 (acc_copy).
```

To learn more, use command: explain ftn-6418

yes, as we expected

Over-cautious: compiler worried about halos; could specify copyout(wrk2)

```
163. 1--------< DO loop = 1,nn
169. 1            gosa1 = 0
171. 1 G-----<> !$acc parallel loop reduction(+:gosa1) private(i,j,k,s0,ss)
172. 1 g------<   DO k = 2,kmax-1
173. 1 g 3----<    DO j = 2,jmax-1
174. 1 g 3 g--<     DO i = 2,imax-1
175. 1 g 3 g         s0 = a(i,j,k,1) * p(i+1,j,k) ...
188. 1 g 3 g-->     ENDDO
189. 1 g 3---->    ENDDO
190. 1 g------>   ENDDO
191. 1          !$acc end parallel loop
208. 1--------> ENDDO

ftn-6430 ftn: ACCEL File = himeno_f77_v02.f, Line = 172
  A loop starting at line 172 was partitioned across the thread blocks.

ftn-6509 ftn: ACCEL File = himeno_f77_v02.f, Line = 173
  A loop starting at line 173 was not partitioned because a better candidate was found at
   line 174.

ftn-6412 ftn: ACCEL File = himeno_f77_v02.f, Line = 173
  A loop starting at line 173 will be redundantly executed.

ftn-6430 ftn: ACCEL File = himeno_f77_v02.f, Line = 174
  A loop starting at line 174 was partitioned across the 128 threads within a threadblock.
```

CUDA: k value(s) built from blockIdx.x

Each thread executes complete j-loop for its i, k value(s)

CUDA: i value(s) built from threadIdx.x

Is the code still correct?

- **Most important thing is that the code is correct:**
  - Make sure you check the residual (Gosa)
  - N.B. will never get bitwise reproducibility between CPU and GPU architectures
    - different compilers will also give different results

- ***Advice: make sure the code has checksums, residuals etc. to check for correctness.***
  - *even if code is single precision, try to use double precision for checking.*
    - *globally or at least for global sums and other reduction variables*

How does this first version perform?

language	Fortran		C	
precision	single	double	single	double
v00	2881	1454	2287	1131
v01	1177	565	1178	594

- **The code is faster...**
  - ... but not by much and compared to one core.
- **Why?**
  - Only 2% of the GPU time is compute;
    - The rest is data transfer to and from device
- ***Lesson: optimise data movements before looking at kernel performance***
  - We are lucky with Himeno
  - most codes are actually slower than one core at this stage

Profiling the first Himeno kernel

```
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% | 11.716 | 11.656 |     23525 |      1680 |      515 |Total
|-----------------------------------------------------------------------------------------------
| 100.0% | 11.716 | 11.656 |     23525 |      1680 |      515 |main_
|        |        |        |           |           |          | jacobi_
3        |        |        |           |           |          |  jacobi_.ACC_REGION@li.288
||||--------------------------------------------------------------------------------------------
4|||   93.5% | 10.953 | 10.911 |     23525 |        -- |      103 |jacobi_.ACC_COPY@li.288
4|||    4.5% |  0.527 |  0.517 |        -- |      1680 |      103 |jacobi_.ACC_COPY@li.315
4|||    2.0% |  0.230 |     -- |        -- |        -- |      103 |jacobi_.ACC_SYNC_WAIT@li.315
4|||    0.0% |  0.004 |  0.228 |        -- |        -- |      103 |jacobi_.ACC_KERNEL@li.288
4|||    0.0% |  0.001 |     -- |        -- |        -- |      103 |jacobi_.ACC_REGION@li.288(exclusive)
|===============================================================================================
```

- **CrayPAT profile, breaks time down into compute and data**
- **Most kernels are launched asynchronously**
  - as is the case with CUDA
  - reported host time is the time taken to launch operation
    - Host time is much smaller than accelerator time
  - Host eventually waits for completion of accelerator operations
    - This shows up in a "large" SYNC_WAIT time

Profiling the first Himeno kernel

```
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% | 11.745 | 11.686 |     23525 |      1680 |      412 |Total
|------------------------------------------------------------------------------------------------
| 100.0% | 11.745 | 11.686 |     23525 |      1680 |      412 |main_
|        |        |        |           |           |          | jacobi_
3        |        |        |           |           |          |  jacobi_.ACC_REGION@li.288
||||--------------------------------------------------------------------------------------------
4|||   93.5% | 10.978 | 10.935 |     23525 |        -- |      103 |jacobi_.ACC_COPY@li.288
4|||    4.5% |  0.532 |  0.523 |        -- |      1680 |      103 |jacobi_.ACC_COPY@li.315
4|||    2.0% |  0.234 |  0.228 |        -- |        -- |      103 |jacobi_.ACC_KERNEL@li.288
4|||    0.0% |  0.001 |     -- |        -- |        -- |      103 |jacobi_.ACC_REGION@li.288(exclusive)
|================================================================================================
```

- **Clarify profile by inserting synchronisation points**
  - Could do this explicitly by inserting "acc wait" after every operation
  - better to compile with CCE using -hacc_model=auto_async_none
    - see man crayftn for details
- **Profile now shows same time for host at every operation**
  - It is now very clear that data transfers take most of the time
- **Extra synchronisation will affect performance**
  - Could skew the profile, so use with care
  - N.B. GPU profilers (Craypat, Nvidia...) already introduce some sync.

Step 3: Optimising data movements

- **Within jacobi routine**
  - data-sloshing: all arrays are copied to GPU at every loop iteration
- **Need to establish data region outside the iteration loop**
  - Then data can remain resident on GPU for entire call
    - reused for each iteration without copying to/from host
  - Must accelerate all loopnests processing the arrays
    - Even if it takes negligible compute time, still accelerate for data locality
      - This is a major productivity win for OpenACC compared to low-level languages
        - You can accelerate a loopnest with one directive
        - Don't have to handcode a new CUDA/OpenCL kernel
        - And, remember, the performance of such a kernel is irrelevant

Step 3: Structure of the jacobi routine

- data region spans iteration loop
  - CPU and OpenACC code
  - use explicit data clauses
    - no automatic scoping
    - requires knowledge of app
  - enclosed kernels shouldn't have data clauses for these variables
  - wrk2 now a scratch array
    - does not need copying

```
SUBROUTINE jacobi(nn,gosa)

!$acc data copy(p)  &
!$acc&     copyin(a,b,c,wrk1,bnd)  &
!$acc&     create(wrk2)
   iteration: DO loop = 1, nn

! compute stencil: wrk2, gosa from p
!$acc parallel loop <clauses>
      <stencil loopnest>
!$acc end parallel loop

! copy back wrk2 into p
!$acc parallel loop
      <copy loopnest>
!$acc end parallel loop

   ENDDO iteration
!$acc end data

END SUBROUTINE jacobi
```

How does this second version perform?

language	Fortran		C	
precision	single	double	single	double
v00	2881	1454	2287	1131
v01	1177	565	1178	594
v02	37525	20300	37143	20287

- **A big performance improvement**
  - Now 51% of the GPU time is compute
    - And more of the profile has been ported to the GPU
  - Data transfers only happen once per call to jacobi(),
    - rather than once per iteration
  - Code still correct:
    - Check the Gosa values

Profile with a local data region in jacobi()

```
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% | 0.497 | 0.475 |   424.177 |    32.630 |     624 |Total
|-----------------------------------------------------------------------------------------
| 100.0% | 0.497 | 0.475 |   424.177 |    32.630 |     624 |main_
|        |       |       |           |           |         | jacobi_
3        |       |       |           |           |         |  jacobi_.ACC_DATA_REGION@li.276
||||-------------------------------------------------------------------------------------
4|||  50.5% | 0.251 | 0.236 |     0.001 |     0.001 |     412 |jacobi_.ACC_REGION@li.288
|||||------------------------------------------------------------------------------------
5||||  46.7% | 0.232 | 0.227 |        -- |        -- |     103 |jacobi_.ACC_KERNEL@li.288
5||||   1.9% | 0.010 | 0.005 |        -- |     0.001 |     103 |jacobi_.ACC_COPY@li.315
5||||   1.8% | 0.009 | 0.004 |     0.001 |        -- |     103 |jacobi_.ACC_COPY@li.288
|||||====================================================================================
4|||  40.0% | 0.199 | 0.197 |   424.176 |        -- |       2 |jacobi_.ACC_COPY@li.276
4|||   7.6% | 0.038 | 0.033 |        -- |        -- |     206 |jacobi_.ACC_REGION@li.317
5|||   7.5% | 0.037 | 0.033 |        -- |        -- |     103 | jacobi_.ACC_KERNEL@li.317
4|||   1.9% | 0.009 | 0.009 |        -- |    32.629 |       2 |jacobi_.ACC_COPY@li.335
|========================================================================================
```

- **Profile now dominated by compute (ACC_KERNEL)**
- **Data transfers infrequent**
  - only once for each of 2 calls to jacobi
  - but still very expensive

Step 4: Further optimising data movements

- **Still including single copy of data arrays in timing of jacobi routine**

- **Solution: move up the call tree to parent routine**
  - Add data region that spans both initialisation and iteration routines
  - Specified arrays then only move on boundaries of outer data region
    - moves the data copies outside of the timed region
      - after all, benchmark aims to measure flops, not PCIe bandwidth

Adding a data region

- Data region spans both calls to jacobi
  - plus timing calls
- Arrays just need to be copyin now
  - and transfers not timed
- Data region remains in jacobi
  - you can nest data regions
  - arrays now declared present
  - could be copy_or_present
  - advice: present generates runtime error if not present
- Drawback: arrays have to be in scope for this to work
  - may need to unpick clever use of module data

```
PROGRAM himeno
   CALL initmt

!$acc data copyin(p,a,b,c,bnd,wrk1,wrk2)
   cpu0 = gettime()
   CALL jacobi(3,gosa)
   cpu1 = gettime()

   cpu0 = gettime()
   CALL jacobi(nn,gosa)
   cpu1 = gettime()
!$acc end data

   END PROGRAM himeno
```

```
SUBROUTINE jacobi(nn,gosa)

!$acc data present(p,a,b,c,wrk1,bnd,wrk2)
   iteration: DO loop = 1, nn

   ENDDO iteration
!$acc end data

END SUBROUTINE jacobi
```

Step 4: Going further

- **Best solution is to port entire application to GPU**
  - data regions span entire use of arrays
  - all enclosed loopnests accelerated with OpenACC
  - no significant data transfers
- **Expand outer data region to include call to initialisation routine**
  - arrays can now all be declared as scratch space with "create"
  - need to accelerated loopnests in initmt(), declaring arrays present
- **N.B. Currently no way to ONLY allocated arrays in GPU memory**
  - CPU version is now dead space, but
  - GPU memory is usually the limiting factor, so usually not a problem.

Porting entire application

- No significant data transfers now
  - doesn't improve measured performance in this case

```
PROGRAM himeno

!$acc data create(p,a,b,c,bnd,wrk1,wrk2)
   CALL initmt
   cpu0 = gettime()
   CALL jacobi(3,gosa)

   CALL jacobi(nn,gosa)
   cpu1 = gettime()
!$acc end data

   END PROGRAM himeno
```

```
SUBROUTINE initmt
!$acc data present(p,a,b,c,wrk1,bnd)
!$acc parallel loop
   <set all elements to zero>

!$acc parallel loop
   <set some elements to be non-zero>
!$acc end data

END SUBROUTINE initmt
```

How does this third version perform?

language	Fortran		C	
precision	single	double	single	double
v00	2881	1454	2287	1131
v01	1177	565	1178	594
v02	37525	20300	37143	20287
v03	51921	28863	51078	28891

- **Code is now a lot faster (44x faster than v01)**
  - 98% of the GPU time is now compute
    - Remaining data transfers are negligible and outside region timed
  - And the code is still correct:
    - Check the Gosa values!
- **We're getting a great speedup: 18x compared to v00**
  - But this is compared to one CPU core out of 16
  - What happens if we use all the cores
    - using OpenMP, as this is originally a scalar code

Profile of fully ported application

```
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% | 0.296 | 0.275 |    0.001 |    0.001 |     634 |Total
|-----------------------------------------------------------------------------------------
| 100.0% | 0.296 | 0.275 |    0.001 |    0.001 |     634 |main_
|        |       |       |          |          |         | main_.ACC_DATA_REGION@li.116
|||----------------------------------------------------------------------------------------
3||  97.6% | 0.289 | 0.269 |    0.001 |    0.001 |     624 |jacobi_
4||        |       |       |          |          |         | jacobi_.ACC_DATA_REGION@li.277
|||||--------------------------------------------------------------------------------------
5|||  84.8% | 0.251 | 0.236 |    0.001 |    0.001 |     412 |jacobi_.ACC_REGION@li.288
||||||-------------------------------------------------------------------------------------
6||||  78.4% | 0.232 | 0.227 |       -- |       -- |     103 |jacobi_.ACC_KERNEL@li.288
6||||   3.3% | 0.010 | 0.005 |       -- |    0.001 |     103 |jacobi_.ACC_COPY@li.315
6||||   3.1% | 0.009 | 0.004 |    0.001 |       -- |     103 |jacobi_.ACC_COPY@li.288
||||||=====================================================================================
5|||  12.7% | 0.038 | 0.033 |       -- |       -- |     206 |jacobi_.ACC_REGION@li.317
6|||  12.7% | 0.038 | 0.033 |       -- |       -- |     103 | jacobi_.ACC_KERNEL@li.317
|||||======================================================================================
3||   1.8% | 0.005 | 0.005 |       -- |       -- |       7 |initmt_
4||        |       |       |          |          |         | initmt_.ACC_DATA_REGION@li.208
|==========================================================================================
```

- **Almost no data transferred**
  - remainder (gosa and a few compiler internals) hard to remove

- **At this point we can start looking at kernel optimisation**

Step 5: Is this a good loop schedule?

- Look at .lst file
- Should see partitioning between and across threadblocks
  - if not, much of GPU is is being wasted

```
172. 1 g------<  DO k = 2,kmax-1
173. 1 g 3----<   DO j = 2,jmax-1
174. 1 g 3 g--<    DO i = 2,imax-1
175. 1 g 3 g        s0 = a(i,j,k,1)*p(i+1,j,k) ...
188. 1 g 3 g-->    ENDDO
189. 1 g 3---->   ENDDO
190. 1 g------>  ENDDO
```

- Usually want inner loop to be vectorised
  - allows coalesced loading of data from global memory
  - if inner loop is not partitioned over threads in a threadblock...
    - is the loop vectorisable (are there dependencies between loop iterations)?
      - No? You need to rewrite the code (it will probably go faster on the CPU)
        - Can you use a more-explicitly parallel algorithm?
        - Avoid incremented counters (e.g. when packing buffers)
        - Change data layout so inner loop addresses fastest-moving array index
      - Yes? You need to tell the compiler what to do:
        - Put "acc loop vector" directive above the "DO i = ..." statement
- This is the most important optimisation
  - almost guaranteed to give big performance increase
  - other optimisations are trial-and-error and may give no benefits

Advanced performance tuning

```
172. 1 g------<  DO k = 2,kmax-1
173. 1 g 3----<   DO j = 2,jmax-1
174. 1 g 3 g--<    DO i = 2,imax-1
175. 1 g 3 g         s0 = a(i,j,k,1)*p(i+1,j,k) ...
188. 1 g 3 g-->    ENDDO
189. 1 g 3---->   ENDDO
190. 1 g------>  ENDDO
```

- Loop schedule balances lots of parallel threads vs. enough work per thread
- If kmax is small, perhaps need more threads
  - Try collapsing k and j loops to get more loop iterations
    - Put "acc loop collapse(2)" directive above k-loop
  - Collapse can be expensive if compiler has to regenerate k and j
    - integer divides are costly
  - Could instead collapse i and j loops, or all three loops
- Nvidia Fermi and Kepler GPUs have caching
  - Loop blocking can improve cache usage (as for the CPU)
    - Block the loops manually (and use gang, vector clauses to tweak schedule)
    - Can use CCE-specific directives to do this as well
- We'll discuss performance optimisation in more detail in a following lecture

Aside: OpenACC and OpenMP (naïve)

```
!$acc data etc.

   iteration: DO loop = 1, nn


      gosa = 0d0

      gosa1 = 0d0
!$acc parallel loop
      reduction(+:gosa1) etc.
      <stencil loopnest>
!$acc end parallel loop

!$acc parallel loop
      <copy loopnest>
!$acc end parallel loop


      gosa = gosa + gosa1

   ENDDO iteration
!$acc end data
```

```
iteration: DO loop = 1, nn


   gosa = 0d0

   gosa1 = 0d0
!$omp parallel do
      reduction(+:gosa1) etc.
      <stencil loopnest>
!$omp end parallel do

!$omp parallel do etc.
      <copy loopnest>
!$omp end parallel do


   gosa = gosa + gosa1

ENDDO iteration
```

Aside: OpenACC and OpenMP (better)

```
!$acc data etc.

   iteration: DO loop = 1, nn



      gosa = 0d0

      gosa1 = 0d0
!$acc parallel loop
      reduction(+:gosa1) etc.
      <stencil loopnest>
!$acc end parallel loop

!$acc parallel loop
      <copy loopnest>
!$acc end parallel loop



      gosa = gosa + gosa1

   ENDDO iteration
!$acc end data
```

```
!$omp parallel
      private(s0,ss,gosa1) etc.
   iteration: DO loop = 1, nn
!$omp barrier
!$omp master
      gosa = 0d0
!$omp end master
      gosa1 = 0d0
!$omp do

      <stencil loopnest>
!$omp end do

!$omp do
      <copy loopnest>
!$omp end do nowait

!$omp critical
      gosa = gosa + gosa1
!$omp end critical
   ENDDO iteration
!$omp end parallel
```

Aside: OpenACC versus OpenMP

- **One OpenMP threadteam for whole iteration loop**
  - OpenMP threads are "heavyweight" (expensive to create teams)
  - need barrier/master/critical regions for synchronisation
  - gosa1 is a thread-private variable; critical region used for reduction

- **Multiple OpenACC parallel loop kernels**
  - GPU kernels execute as threadblocks on different SMs
    - need separate kernels to synchronise
      - no global synchronisation method within a kernel
  - GPU threads are "lightweight" (kernels cheap to launch)
  - gosa1 is a shared reduction loop variable

- **Can we use a single OpenACC parallel region?**
  - Containing scalar code and multiple loop nests
  - Analogous to OpenMP parallel region
  - **NO!** Race conditions between kernels and with scalar code

Results using OpenMP across the CPU

language	Fortran		C	
precision	single	double	single	double
v00	2881	1454	2287	1131
v01	1177	565	1178	594
v02	37525	20300	37143	20287
v03	51921	28863	51078	28891
OMP16	8996	5416	9761	5086

- **Fastest OpenMP version uses 16 threads**
  - but only 3-4x faster than serial version
  - (code is quite memory bandwidth bound)

- **Overall speedup: 5-6x**
  - This is the sort of figure we expect
    - Kepler GPU memory bandwidth around 6x compared to Interlagos CPU

In summary

- **We ported the entire Himeno code to the GPU**
  - chiefly to avoid data transfers
    - 4 OpenACC kernels (only 1 significant for compute performance)
    - 1 outer data region
    - 2 inner data regions (nested within this)
  - 7 directive pairs for 200 lines of Fortran
  - Profiling frequently showed the bottlenecks
  - Correctness was also frequently checked
- **Data transfers were optimised at the first step**
- **We checked the kernels were scheduling sensibly**
- **Further performance tuning**
  - data region gave a 44x speedup; kernel tuning is secondary
  - Low-level languages like CUDA offer more direct control of the hardware
    - OpenACC is much easier to use, and should get close to CUDA performance
  - Remember Amdahl's Law:
    - speed up the compute of a parallel application, soon become network bound
    - Don't waste time trying to get an extra 10% in the compute
    - You are better concentrating your efforts on tuning the MPI/CAF comms
- **Bottom line:**
  - 5-6x speedup from 7 directive pairs in 200 lines of Fortran
  - compared to the complete CPU

Now it's your turn

- **In the next practical, you will go through the steps of accelerating this same code**

- **Your learning outcomes:**
  - You should:
    - Check that you understand how to OpenACC a simple code
      - Either edit v00 to add the directives yourself
      - Or read the prepared v01, v02, v03 files and see it makes sense
    - Verify the results that I quote here for CCE
  - You could:
    - Try generating some profiles using CrayPAT