First work example: the scalar Himeno code

Mandes Schönherr

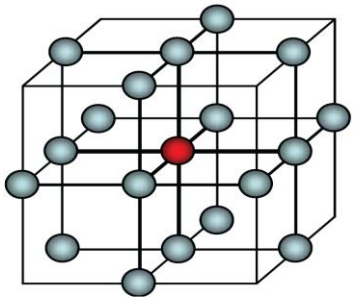
Overview

- **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

- **Beside the theoretical background, we will implement the OpenMP directives and compare results.**

The Himeno Benchmark

- **3D Poisson equation solver**
  - Iterative loop evaluating 19-point stencil
  - Memory intensive, memory bandwidth bound



- **Fortran and C implementations available**

- **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* OpenMP an application**
  - i.e. one that runs faster node-for-node
  - not just full accelerator vs. single CPU core

- **There are two key things to understand about the code:**
  - How is data passed through the calltree?
  - Where are the hotspots?

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

The key questions in detail

- **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 OpenMP/OpenACC port will keep data resident on the accelerator

- **Where are the hotspots?**
  - The OpenMP/OpenACC programming model is aimed at loop-based codes
    - Which loopnests dominate the runtime?
    - Are they suitable for an accelerator?
      - What are the min/average/max tripcounts?

- **Minimising data movements will probably require acceleration of many more (and possibly all) loopnests**
  - Not just the hotspots
    - any loopnest that processes arrays that we want accelerator-resident
  - But we have to start somewhere

First stages to accelerating an application

Understand and characterise the application

- Profiling tools, code inspection, speaking to developers if you can

1. Introduce first OpenMP kernels

2. Introduce data regions in subprograms

- reduce unnecessary data movements
- will probably require more OpenMP kernels

Next stages to accelerating an application

3. **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

4. **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 OpenMP tuning clauses
     - You may gain some final additional performance from this

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

We will treat in another session

Initial run

Reference

Understand the application

Run the initial CPU version of the code

- **First compile and run the code "as is“**
  - on the CPU, using one core
  - including a performance profile

- **compiled with Cray Compilation Environment (CCE):**

```
$> module switch / load …
$> module load perftools-base
$> module load perftools-lite-loops
$> make VERSION=00
$> sbatch submit.wlm
```

The application output :

- The runtime

```
Time (secs) : 3.4616304183145985
```

- The performance
  (from the runtime, in MFLOPS)

```
MFLOPS      : 3960.7098225914551
```

- A residual value from the solver
  (checksum)

```
Gosa  : 0.13783597243842747 9E-02
```

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*

Profile of the initial CPU version of the code

- **most expensive computational computation**
  - In loop at line 232
  - when suitable, accelerate this operation

```
$> cat himeno_v00.log
...
Table 1:  Inclusive and Exclusive Time in Loops (from -hprofile_generate)

  Loop |     Loop |     Time |  Loop Hit |  Loop |  Loop |  Loop |Function=/.LOOP[.]
  Incl |     Incl |    (Loop |           | Trips | Trips | Trips |
 Time% |     Time |    Adj.) |           |   Avg |   Min |   Max |
|-------------------------------------------------------------------------------
| 95.9% | 3.576932 | 0.000014 |         2 |  51.5 |     3 |   100 |jacobi_.LOOP.1.li.263
| 82.6% | 3.080879 | 0.000533 |       103 | 126.0 |   126 |   126 |jacobi_.LOOP.2.li.270
| 82.6% | 3.080346 | 0.049932 |    12,978 | 126.0 |   126 |   126 |jacobi_.LOOP.3.li.271
```

- **Let’s have a closer look to the code**

Himeno program structure

Initialisation routine

Stencil routine

Calibration run

Measurement run

```
PROGRAM himeno       ! EDITED HIGHLIGHTS ONLY
  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 *,' Time:  ',cpu
  PRINT *,' Gosa:  ',gosa
  PRINT *,' MFLOPS:',xmflops2
END PROGRAM himeno
```

- It's useful to hardwire nn
  - gosa is then a checksum
- xmflops2 shows performance
- Now we look at the details of jacobi()

Structure of the jacobi routine

- **Iteration loop**
  - must be sequential!
- **Apply stencil to `p`**
  - create temporary `wrk2`
  - residual `gosa` computed
- **Update array `p`**
  - from `wrk2`
  - can be parallelised
  - outer halo unchanged

```
SUBROUTINE jacobi(nn,gosa)

  iter_lp: 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 iter_lp

END SUBROUTINE jacobi
```

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 is computed
- 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.

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 map(*)
    - 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

- **Generate listing files**
  - CCE:

```
FLAGS+=-hlist=a
```

Produces commentary files <stem>.lst

  - PGI:

```
FLAGS+=-Minfo
```

Feedback to STDERR

  - Compiler teams work very hard to make feedback useful
  - advice: use it, it's free!
    (i.e. no impact on performance)

Compiler listings

source line numbers

+ = additional message below

Numbers denote serial loops

b = blocked loops

V = vectorized

r = unrolled

```
263.  + 1------------<    iter_loop: DO loop = 1,nn
. . .
270.  + 1 b----------<     DO k = 2,kmax-1
271.  + 1 b b--------<      DO j = 2,jmax-1
272     1 b b Vr3----<       DO i = 2,imax-1
273.    1 b b Vr3             S0 = a(i,j,k,1)* ...
. . .
289.    1 b b Vr3             wrk2(i,j,k)=p(i,j,k)+…
290.    1 b b Vr3---->       ENDDO
291.    1 b b-------->      ENDDO
292.    1 b---------->     ENDDO
. . .
306.    1------------>    ENDDO iter_loop
```

First kernel

Now we know where,
so we implement it and compare.

Versioning

- **Create a new version**
  - Thus we can compare
  - We can go back, when we messed up

- **Here we simply create a copy of the code**

```
$> cp himeno_*_v00.* himeno_*_v01.*
```

Step 1: a first OpenMP device kernel

- Start with most expensive
  - apply teams distribute
- reduction clause
  - Necessary for correct gathering of data
- private clause (optional here)
  - By default:
    - loop variables private (i, j, k)
    - scalar variables are private (s0,ss)
  - Note that private **arrays**
    - always need private clause

```
gosa1 = 0d0
!$omp target teams distribute &
!$omp& reduction(+:gosa1) &
!$omp& private(i,j,k,s0,ss)
DO k = 2,kmax-1
 DO j = 2,jmax-1
  DO i = 2,imax-1
   s0 = a(i,j,k,1) * p(i+1,j, k ) &
     <etc...>
   ss = (s0*a(i,j,k,4)-p(i,j,k))* &
        bnd(i,j,k)
   gosa1 = gosa1 + ss*ss
   wrk2(i,j,k) = p(i,j,k)+omega*ss
  ENDDO
 ENDDO
ENDDO
!$omp end target teams distribute
```

Compiler listings: `vi himeno_*_v01.lst`

- Check what the compiler did

G = accelerated

g = partioned

b = blocked loops

```
269.  + 1 G----------< !$omp target teams distribute reduc...
270.    1 G g--------<     DO k = 2,kmax-1
271.  + 1 G g b------<      DO j = 2,jmax-1
272.    1 G g b gb---<       DO i = 2,imax-1
273.    1 G g b gb            S0 = a(i,j,k,1)*p(i+1,j,k) &
. . .
287.    1 G g b gb            ss = (s0*a(i,j,k,4)-p(i,j,k))...
. . .
288.    1 G g b gb            gosa1 = gosa1 + ss*ss
289.    1 G g b gb            wrk2(i,j,k) = p(i,j,k)+omega*ss
290.    1 G g b gb--->       ENDDO
291.    1 G g b------>      ENDDO
292.    1 G g-------->     ENDDO
293.    1 G----------> !$omp end target teams distribute
```

Compiler listings

- `vi himeno_*_v01.lst`
- Below each function are additional annotations

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

```
  269.  + 1 G----------< !$omp target teams distribute reduc...
  270.    1 G g--------<     DO k = 2,kmax-1
  271.  + 1 G g b------<      DO j = 2,jmax-1
  272.    1 G g b gb---<       DO i = 2,imax-1
  273.    1 G g b gb            S0 = a(i,j,k,1)*p(i+1,j,k) &
  . . .
  290.    1 G g b gb--->       ENDDO
  291.    1 G g b------>      ENDDO
  292.    1 G g-------->     ENDDO
  293.    1 G----------> !$omp end target teams distribute
ftn-6405 ftn: ACCEL File = himeno_F_v01.F90, Line = 269
  A region starting at line 269 and ending at line 293 was placed on
    the accelerator.
ftn-6430 ftn: ACCEL File = himeno_F_v01.F90, Line = 270
  A loop starting at line 270 was partitioned across the thread
    blocks.
ftn-6412 ftn: ACCEL File = himeno_F_v01.F90, Line = 271
  A loop starting at line 271 will be redundantly executed.
ftn-6430 ftn: ACCEL File = himeno_F_v01.F90, Line = 272
  A loop starting at line 272 was partitioned across the 128 threads
    within a threadblock.
```

Performance with the first kernel

- **Original performance:**

```
$> make VERSION=00
$> qsub submit.wlm
```

```
Gosa          : 0.13783597243844302 7E-02
MFLOPS         :  3750.2014219750636
```

- **With one OpenMP device kernel:**

```
$> module load craype-accel-nvidia60
$> module switch perftools-lite-loops perftools-lite-gpu
$> make VERSION=01 OMP=yes
$> vi submit.wlm # change to v01
$> sbatch submit.wlm
```

Now we use the gpu version

```
Gosa          : 0.13783597243839072 5E-02
MFLOPS         :  2413.6436835812342
```

- **The code is slower**
  - Why?
  - How do we find out?

more Compiler listings

To learn more, use command:
explain ftn-6418

yes, as we expected

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

```
269.  + 1 G----------< !$omp target teams distribute reduc...
270.    1 G g--------<     DO k = 2,kmax-1
271.  + 1 G g b------<      DO j = 2,jmax-1
272.    1 G g b gb---<       DO i = 2,imax-1
273.    1 G g b gb            S0 = a(i,j,k,1)*p(i+1,j,k) &
. . .
290.    1 G g b gb--->       ENDDO
291.    1 G g b------>      ENDDO
292.    1 G g-------->     ENDDO
293.    1 G----------> !$omp end target teams distribute
Data movements:
ftn-6418 ftn: ACCEL File = himeno_F_v01.F90, Line = 269
  If not already present: allocate memory and copy whole array "p" to
    accelerator, free at line 293 (acc_copyin).
<identical messages for a,b,c,wrk1,bnd>

ftn-6416 ftn: ACCEL File = himeno_F_v01.F90, Line = 269
  If not already present: allocate memory and copy whole array "wrk2"
    to accelerator, copy back at line 293 (acc_copy).
```

Step 1a: data scoping

- data map clauses
  - compiler will do automatic analysis
    - usually correct
    - but can be over-cautious
- ***advice:***
  - *only use clauses if compiler over-cautious*
  - explicit data clauses will interfere with data directives at next step

```
gosa1 = 0d0
!$omp target teams distribute &
!$omp&   reduction(+:gosa1) &
!$omp&   private(i,j,k,so,ss) &  #optional
!$omp&   map(to:a,b,c,bnd,wrk1) &
!$omp&   map(to:p) map(from:wrk2)
DO k = 2,kmax-1
 DO j = 2,jmax-1
  DO i = 2,imax-1
   s0 = a(i,j,k,1) * p(i+1,j, k ) &
     <etc...>
     ss = (s0*a(i,j,k,4) - p(i,j,k)) * &
           bnd(i,j,k)
   gosa1 = gosa1 + ss*ss
   wrk2(i,j,k) = p(i,j,k) + omega*ss
  ENDDO
 ENDDO
ENDDO
!$omp end target teams distribute
```

Performance with the first kernel

- **Original performance:**

```
Gosa        : 0.13783597243844302 7E-02
MFLOPS       :  3750.2014219750636
```

- **With one OpenMP device kernel:**

```
$> make VERSION=01a OMP=yes
$> vi submit.wlm # change to v01a
$> sbatch submit.wlm
```

```
Gosa        : 0.13783597243839072 5E-02
MFLOPS       :  2568.1106308604203
```

- **The code is still slower**
  - Why?
  - How do we find out?

Searching for performance issues

- **First we look at the compiler feedback**
  - Are the loops scheduling sensibly?
  - We could have a look again, but the answer is "yes".
- **Next, we look at the runtime commentary**
  - An event-by-event record of application execution
  - Tools from Cray and NVIDIA
- **Cray runtime commentary:**
  - Compile with CCE (no special options)
  - Set environment variable:

    `CRAY_ACC_DEBUG=2`
  - **Re-run** the executable
  - The commentary is written to STDERR

Cray runtime commentary

- CRAY_ACC_DEBUG=2
- **Copy in/out is observed for every iteration step**

Allocation and copy data to the accelerator

Copy back and release data from the accelerator

large arrays

- **How often we do that?**
  - Could we reduce it?

```
ACC: Start transfer 16 items from himeno_F_v01a.F90:153
ACC:        allocate, copy to acc 'a' (136855584 bytes)
ACC:        allocate, copy to acc 'b' (102641688 bytes)
ACC:        allocate, copy to acc 'bnd' (34213896 bytes)
ACC:        allocate, copy to acc 'c' (102641688 bytes)
ACC:        allocate, copy to acc 'gosa1' (8 bytes)
ACC:        allocate, copy to acc 'imax' (4 bytes)
. . .
ACC:        allocate 'wrk2' (34213896 bytes)
ACC:        allocate reusable, copy to acc <internal> (4 bytes)
ACC:        allocate reusable <internal> (1008 bytes)
ACC: End transfer (to acc 444780680 bytes, to host 0 bytes)
ACC: Execute kernel jacobi_$rk_L153_1 blocks:126 threads:
             128 async(auto) from himeno_F_v01a.F90:153
ACC: Wait async(auto) from himeno_F_v01a.F90:153
ACC: Start transfer 16 items from himeno_F_v01a.F90:153
ACC:        free 'a' (136855584 bytes)
ACC:        free 'b' (102641688 bytes)
ACC:        free 'bnd' (34213896 bytes)
ACC:        free 'c' (102641688 bytes)
ACC:        copy to host, free 'gosa1' (8 bytes)
ACC:        free 'imax' (4 bytes)
. . .
ACC:        copy to host, free 'wrk2' (34213896 bytes)
ACC:        done reusable <internal> (4 bytes)
ACC:        done reusable <internal> (0 bytes)
ACC: End transfer (to acc 0 bytes, to host 34213904 bytes)
```

Searching for performance issues (II)

- **NVIDIA Compute Profiler:**
  - Event-by-event timing information
  - Compile with CCE (no special options)
  - Set environment variable: `COMPUTE_PROFILE=1`
  - Run the executable
  - The commentary is written to file cuda_profile_0.log

- **Advice:**
  - Don't set both: `CRAY_ACC_DEBUG` with `COMPUTE_PROFILE`

large arrays, need long to copy

every iteration step

```
$> cat cuda_profile_0.log
. . .
method=[ memcpyHtoD ] gputime=[ 22636.031 ] cputime=[ 22714.756 ]
method=[ memcpyHtoD ] gputime=[ 16801.504 ] cputime=[ 16814.359 ]
method=[ memcpyHtoD ] gputime=[ 5632.544 ] cputime=[ 5643.533 ]
method=[ memcpyHtoD ] gputime=[ 16837.633 ] cputime=[ 16847.324 ]
method=[ memcpyHtoD ] gputime=[ 1.088 ] cputime=[ 7.279 ]
. . .
method=[ jacobi_$ck_L153_1 ] gputime=[ 3737.728 ]
            cputime=[ 15.326 ] occupancy=[ 0.312 ]
method=[ memcpyDtoH ] gputime=[ 2.400 ] cputime=[ 19.159 ]
method=[ memcpyDtoH ] gputime=[ 15007.552 ] cputime=[ 15675.952 ]
```

Step 2: Optimising data movements

- **Within jacobi routine**
  - data-sloshing: all arrays are copied to GPU and back 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
    - Must still accelerate for data locality
      - This can be a lot of work
        - Performance of the kernels is irrelevant
      - A major productivity win for OpenMP compared to low-level languages
        - You can accelerate a loopnest with one directive; usually no need for tuning
        - You don't have to handcode a new CUDA kernel

Step 2: Structure of the jacobi routine

- data region spans iteration loop
  - includes both CPU and accelerator code
  - need explicit data clauses
    - no automatic scoping
    - requires knowledge of app
  - **wrk2** now a scratch array
    - does not need copying

```
SUBROUTINE jacobi(nn,gosa)
. . .
!$omp target data map(tofrom:p) &
!$omp&  map(to:a,b,c,wrk1,bnd) &
!$omp&  map(alloc:wrk2)
   iter_lp: DO loop = 1,nn
      gosa = 0d0
!$omp target teams distribute <clauses>
      ! compute stencil: wrk2, gosa from p
      <stencil loopnest>
!$omp end target teams distribute
!$omp target teams distribute
     ! copy back wrk2 into p
      <copy loopnest>
!$omp end target teams distribute
   ENDDO iter_lp
!$omp end target data
END SUBROUTINE jacobi
```

`map(alloc:wrk2)` instead of `map(from:wrk2)`

Additional kernel to proceed everything on accelerator

Step 2: Performance with the first kernel

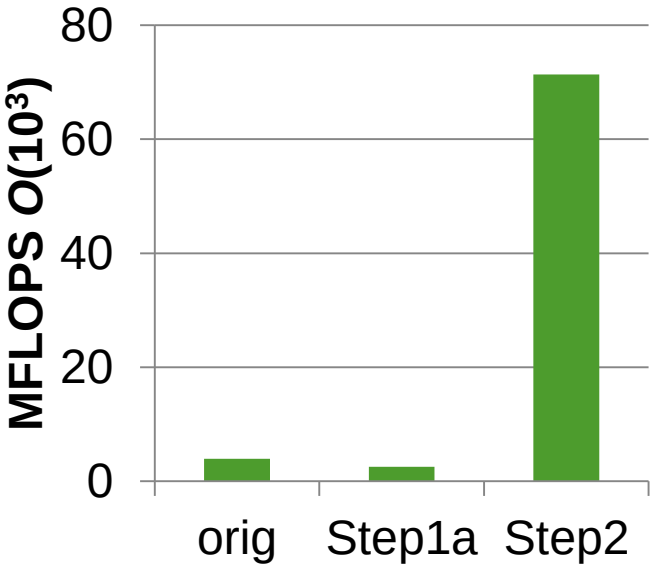
- **Original performance:**

```
Gosa        : 0.137835972438443027E-02
MFLOPS       :  3750.2014219750636
```

- **With one OpenMP data region:**

```
$> make VERSION=02 OMP=yes
$> vi submit.wlm # change to v02
$> sbatch submit.wlm
```

```
Gosa        : 0.137835972438390725E-02
MFLOPS       :  71339.340223725318
```



CRAY_ACC_DEBUG=2

```
<Start of data region>
ACC: Start transfer 7 items from himeno_F_v02.F90:276
ACC:        allocate, copy to acc 'a' (136855584 bytes)
ACC:        allocate, copy to acc 'b' (102641688 bytes)
ACC:        allocate, copy to acc 'bnd' (34213896 bytes)
ACC:        allocate, copy to acc 'c' (102641688 bytes)
ACC:        allocate, copy to acc 'p' (34213896 bytes)
ACC:        allocate, copy to acc 'wrk1' (34213896 bytes)
ACC:        allocate 'wrk2' (34213896 bytes)
ACC: End transfer (to acc 444780648 bytes, to host 0 bytes)

<For each loop iteration... (see next slide)>

<After iteration loop finishes, close of data region>
ACC: Wait async(auto) from himeno_F_v02.F90:329
ACC: Start transfer 7 items from himeno_F_v02.F90:329
ACC:        free 'a' (136855584 bytes)
ACC:        free 'b' (102641688 bytes)
ACC:        free 'bnd' (34213896 bytes)
ACC:        free 'c' (102641688 bytes)
ACC:        copy to host, free 'p' (34213896 bytes)
ACC:        free 'wrk1' (34213896 bytes)
ACC:        free 'wrk2' (34213896 bytes)
ACC: End transfer (to acc 0 bytes, to host 34213896 byt
```

Huge data movement at beginning and end of data region

```
<each iteration loop contains e.g.>
. . .
ACC: Wait async(auto) from himeno_F_v02.F90:311
ACC: Start transfer 7 items from himeno_F_v02.F90:311
ACC:        copy to host, free 'gosa1' (8 bytes)
ACC:        free 'imax' (4 bytes)
ACC:        free 'jmax' (4 bytes)
ACC:        free 'kmax' (4 bytes)
ACC:        free 'omega' (8 bytes)
ACC:        done reusable <internal> (4 bytes)
ACC:        done reusable <internal> (0 bytes)
ACC: End transfer (to acc 0 bytes, to host 8 bytes)
ACC: Start transfer 3 items from himeno_F_v02.F90:313
ACC:        allocate, copy to acc 'imax' (4 bytes)
ACC:        allocate, copy to acc 'jmax' (4 bytes)
ACC:        allocate, copy to acc 'kmax' (4 bytes)
ACC: End transfer (to acc 12 bytes, to host 0 bytes)
. . .
```

Almost no data movement during iterations

In summary

- **We ported the parts of the Himeno code to the GPU**
  - Add data region to avoid data transfers
    - 2 OpenACC kernels (only 1 significant for compute performance)
    - 1 data region
  - 3 directive pairs for 70 lines of Fortran/C
  - Correctness was also frequently checked

- **First step was localizing suitable kernels**
- **First optimization of data movement**

- **Next steps**
  - Profiling application and accelerator data
  - Checking kernels are scheduling sensibly
  - Look at kernel optimisation