First work example: the accelerated scalar Himeno code

Having a more detailed view

Mandes Schönherr

Profiling the first steps

- Comparing versions 01a and 02

```
$> module load craype-accel-nvidia60
$> module load perftools-base
$> module load perftools-lite-gpu
$> make VERSION=XX OMP=yes
$> sbatch submit.wlm  # if necessary change to himeno_vXX.x
```

Why is version 2 so much faster?

- v01a: Single kernel

```
Program invocation:  himeno_v01a.x
Table 1:  Accelerator Table by Function (top 10 functions shown)

   Host | Host |  Acc | Acc Copy | Acc Copy | Events |Function=[max10]
  Time% | Time | Time |       In |      Out |        | PE=HIDE
        |      |      | (MBytes) | (MBytes) |        |  Thread=HIDE

 100.0% | 0.06 | 8.68 |   43,690 |    3,361 |    515 |Total
|-------------------------------------------------------------------------------
|  88.5% | 0.05 | 8.04 |   42,418 |    3,263 |    200 |main_.ACC_COPY@li.174
|   3.9% | 0.00 | 0.38 |       -- |       -- |    100 |main_.ACC_ASYNC_KERNEL@li.174
|   3.3% | 0.00 | 0.25 |    1,273 |    97.89 |      6 |main_.ACC_COPY@li.153
|   2.2% | 0.00 |   -- |       -- |       -- |    100 |main_.ACC_SYNC_WAIT@li.174
|   1.7% | 0.00 |   -- |       -- |       -- |    100 |main_.ACC_REGION@li.174
|===============================================================================
```

- v02: data region in jacobi()

```
Program invocation:  himeno_v02.x
Table 1:  Accelerator Table by Function (top 10 functions shown)

   Host | Host |  Acc | Acc Copy | Acc Copy | Events |Function=[max10]
  Time% | Time | Time |       In |      Out |        | PE=HIDE
        |      |      | (MBytes) | (MBytes) |        |  Thread=HIDE

 100.0% | 0.03 | 0.61 |   848.36 |    65.26 |    830 |Total
|-------------------------------------------------------------------------------
|  70.8% | 0.02 | 0.10 |   424.18 |    32.63 |    402 |main_.ACC_COPY@li.174
|  21.2% | 0.01 | 0.42 |       -- |       -- |    200 |main_.ACC_KERNEL@li.174
|   5.0% | 0.00 | 0.08 |   424.18 |    32.63 |     14 |main_.ACC_COPY@li.153
|   1.6% | 0.00 |   -- |       -- |       -- |    200 |main_.ACC_REGION@li.174
|===============================================================================
```

reduced data copy

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

Already done

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

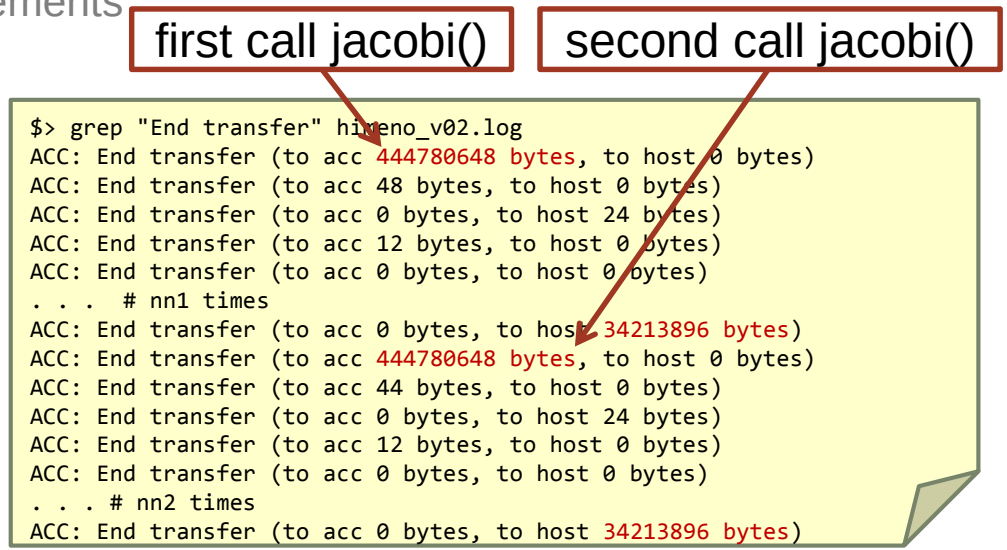
This session

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...**

Step 3: Further optimising data movements

- The code times the calls to jacobi() routine
- Each call contains a data region
  - So we are still timing some data movements
- Solution: move up the call tree to parent routine
  - Add a second, outer data region
    - Spans calls to iteration routines
  - Specified arrays then only move on boundaries of outer data region
  - moves the data copies outside of the timed region



Step3: Adding a data region

- Spans both calls to jacobi
- Arrays just copyin now
  - and transfers not timed
  - `p` can be to
  - `wrk2` can be alloc
- data region in jacobi()
  - not necessary
  - but you keep it (nest data regions)
  - if present data is not touched
- Drawback: arrays have to be in scope for this to work
  - may need to unpick clever use of module data
  - or use OpenMP4.5 unstructured data regions

```
PROGRAM himeno
    CALL initmt
!$omp target data &
!$omp& map(to:a,b,c,bnd,wrk1)&
!$omp& map(to:p) map(alloc:wrk2)
    cpu0 = gettime()
    CALL jacobi(3,gosa)
    cpu1 = gettime()

    cpu0 = gettime()
    CALL jacobi(nn,gosa)
    cpu1 = gettime()
!$omp end target data
END PROGRAM himeno
```

outer data region

p is defined as copyin

data clauses here optional

if jacobi() is called outside the outer data region, data movement is defined

```
SUBROUTINE jacobi(nn,gosa)
!$omp target data map(tofrom:p)&
!$omp& map(to:a,b,c,bnd,wrk1)&
!$omp& map(alloc:wrk2)
   iter_lp: DO loop = 1,nn
      <...>
   ENDDO iter_lp
!$omp end target data
END SUBROUTINE jacobi
```

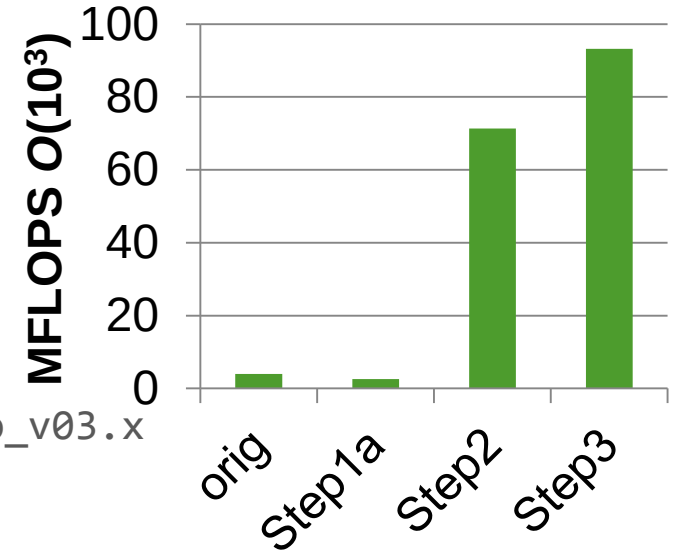
Step 3: Performance with the first kernel

- **Original performance:**

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

- **With outer OpenMP data region:**

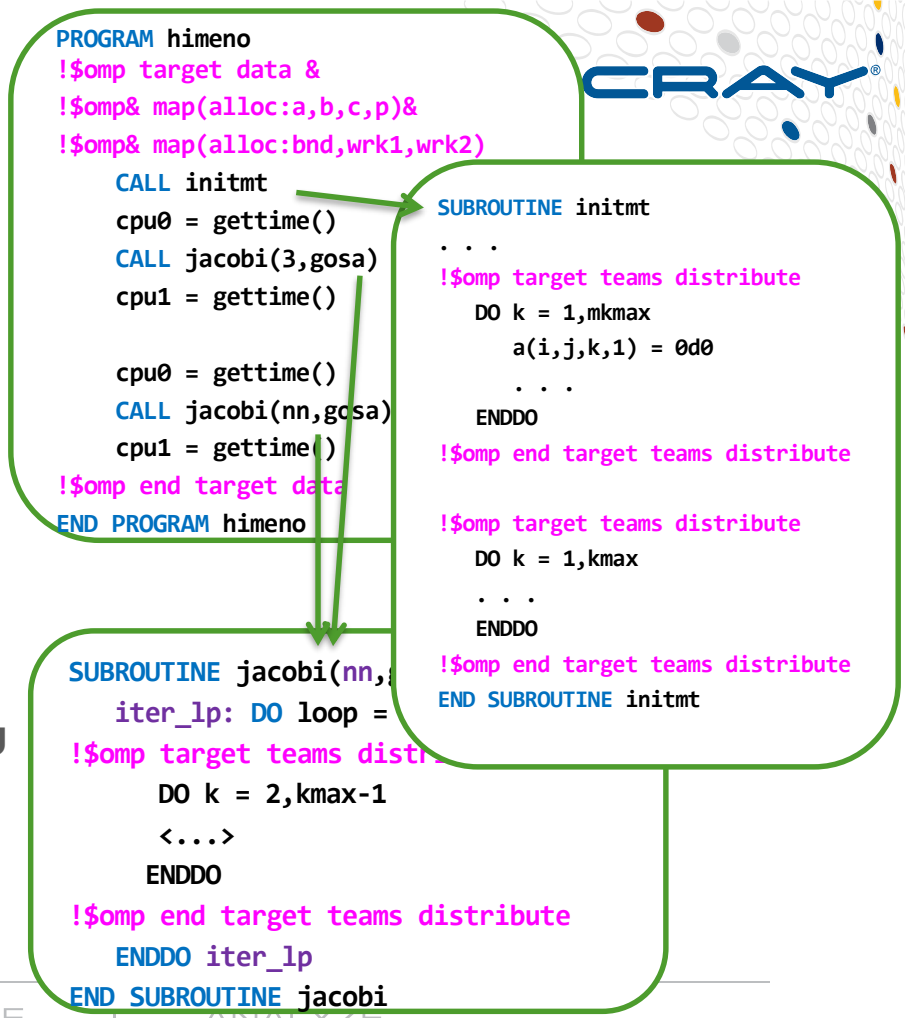
```
$> make VERSION=03 OMP=yes
$> sbatch submit.wlm # change to himeno_v03.x
```




```
 Gosa        : 0.13783597243839072 5E-02
MFLOPS       :  93190.630878365104
```

Step 3a: Going further

- **Best solution is to port entire application to GPU**
  - data regions span entire use of arrays
  - all enclosed loopnests accelerated with OpenMP device constructs
  - no significant data transfers
- **Expand outer data region**
  - to include call to initialisation routine as well
  - arrays can now all be declared as scratch space with alloc clause
  - need to accelerated loopnests in initmt(), no data scoping risk over-cautious
- **N.B. No easy way to ONLY allocate arrays in GPU memory**
  - CPU version is now dead space, but
  - GPU memory is usually the limiting factor, so usually not a problem



Step 3a: Performance with the first kernel

- **Original performance:**

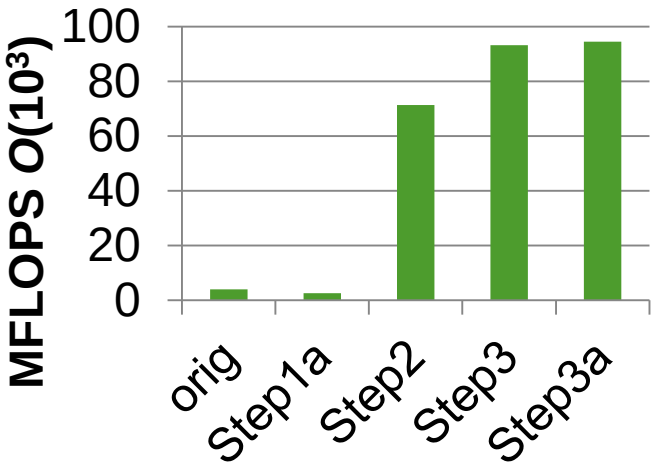
```
 Gosa        : 0.13783597243844302 7E-02
 MFLOPS      :  3750.2014219750636
```

- **whole main in OpenMP data region:**

```
$> make VERSION=03a OMP=yes
$> sbatch submit.wlm # change to himeno_v03a.x
```

```
 Gosa        : 0.13783597243839642 8E-02
MFLOPS       :  94522.234347635153
```

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



Step 4: Performance with larger problem size

- **Original performance:**

```
$> make VERSION=00 OMP=yes \
     PROBLEM_SIZE=4
$> sbatch submit.wlm # change to himeno_v00.x
```

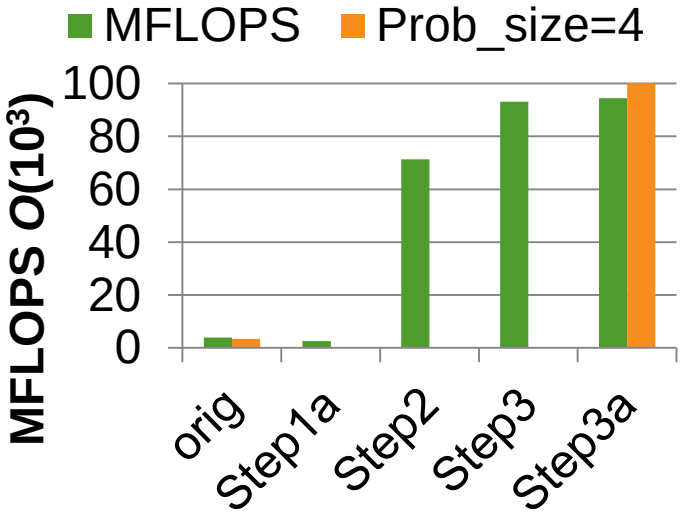
```
Gosa        : 0.775850492248915911E-03
MFLOPS      :  3368.2488685152798
```

- **whole main in OpenMP data region:**

```
$> make VERSION=03a OMP=yes \
     PROBLEM_SIZE=4
$> sbatch submit.wlm # change to himeno_v03a.x
```

```
 Gosa        : 0.775850492248780278E-03
MFLOPS      :  100140.70427375865
```

- Why does this bigger problem perform better?



Now it's your turn

- **Try further optimizations**
  - vary problem size
  - advanced loop scheduling
    - tuning clauses
    - vary threads per block
    - loop collapsing

In summary

- **We ported the entire Himeno code to the GPU**
  - chiefly to avoid data transfers
    - 4 OpenMP kernels (only 1 significant for compute performance)
    - 1 outer data region
    - 1 inner data regions (nested within this)
  - 6 directive pairs for 200 lines of Fortran/C
  - Profiling frequently showed the bottlenecks
  - Correctness was also frequently checked

- **First step was optimising data transfers**

- **Next steps**
  - Checking kernels are scheduling sensibly
  - Look at kernel optimisation

In summary... continued

- **Further performance tuning**
  - data region gave a 22x speedup; kernel tuning is secondary
  - Low-level languages like CUDA
    - offer more direct control of the hardware
    - but OpenMP is much easier to use
      - should get close to CUDA performance
  - Remember Amdahl's Law:
    - speed up the compute of a parallel application,
      - and 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 comms or I/O

- **Bottom line:**
  - 3-4x speedup from 7 directive pairs in 200 lines of Fortran/C
    - performance comparing GPU to the complete CPU