Advanced OpenACC: topics and performance tuning

Alistair Hart
Cray Exascale Research Initiative Europe

Contents

- **Some more advanced OpenACC topics**
  - the async and cache clauses
- **Then we talk about a few tuning tips for OpenACC**
  - The Golden Rules of Tuning
    - information sources
  - Tuning data locality
  - Tuning kernels
    - correcting obvious scheduling errors
    - advanced schedule tuning (collapse, worker, vector_length clauses)
      - use scalar Himeno code as an example
  - Extreme tuning
    - source code changes, reordering data structures, using CUDA
  - Where to learn more

OpenACC async clause

- **async[(handle)] clause for parallel, update directives**
  - Launch accelerator region/data transfer asynchronously
  - Operations with same handle guaranteed to execute sequentially
    - as for CUDA streams
  - Operations with different handles can overlap
    - if the hardware permits it and runtime chooses to schedule it:
    - can potentially overlap:
      - PCIe transfers in both directions
      - Plus multiple kernels
    - can overlap up to 16 parallel streams with Fermi
  - streams identified by handle (integer-valued)
    - tasks with same handle execute sequentially
    - can wait on one, more or all tasks

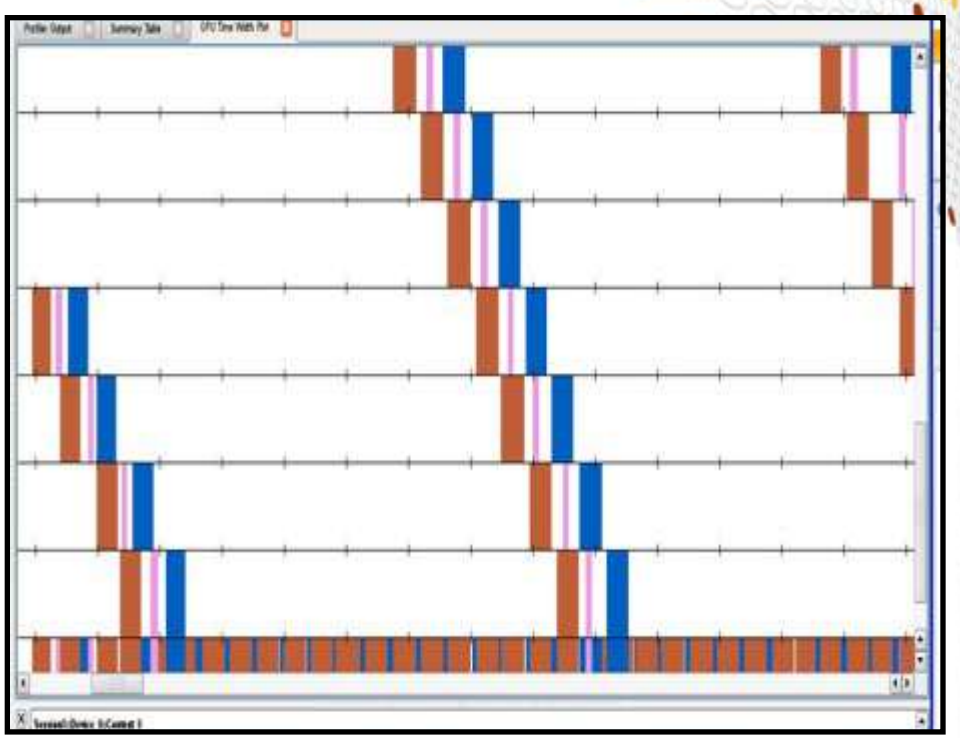
- **!$acc wait: waits for completion of all streams of tasks**
  - !$acc wait(handle) waits for a specified stream to complete
- **Runtime API library functions**
  - can also be used to wait or test for completion

OpenACC async clause

- **First attempt**
  - a simple pipeline:
  - processes array, slice by slice
    - copy data to GPU,
    - process on GPU,
    - bring back to CPU
  - can overlap 3 streams at once
    - use slice number as stream handle
      - don't worry if number gets too large
      - OpenACC runtime maps it back into allowable range (using MOD function)

```
REAL(kind=dp) ::
a(Nvec,Nchunks),b(Nvec,Nchunks)

!$acc data create(a,b)
DO j = 1,Nchunks
!$acc update device(a(:,j)) async(j)

!$acc parallel loop async(j)
  DO i = 1,Nvec
    b(i,j) = <function of a(i,j)>
  ENDDO

!$acc update host(b(:,j)) async(j)

ENDDO
!$acc wait
!$acc end data
```

OpenACC async results



- **Execution times (on Cray XK6):**
  - CPU: 3.76s
  - OpenACC, blocking: 1.10s
  - OpenACC, async: 0.34s
- **NVIDIA Visual profiler:**
  - time flows left to right
  - streams stacked vertically
    - only 7 of 16 streams fit in window
    - **red**: data transfer to GPU
    - **pink**: computational on GPU
    - **blue**: data transfer from GPU
  - vertical slice shows what is overlapping
    - collapsed view at bottom
  - async handle modded by number of streams
    - so see multiple coloured bars per stream (looking horizontally)
- **Alternative to pipelining is task-based overlap**
  - Harder to arrange; needs knowledge of data flow in specific application
  - May (probably will) require application restructuring (maybe helps CPU)
  - Some results later in Himeno Case Study

Using the cache clause

- **Performance-tuning clause**
  - Don't worry about this when first accelerating a code
  - Apply it later to the slowest kernels of working OpenACC port
- **Suggests that compiler could place data into software-managed cache**
  - e.g. threadblock-specific "shared" memory on Nvidia GPU
  - No guarantee it makes the code faster
    - could conflict with automatic caching done by hardware and/or runtime
- **Clause inserted inside kernel**
  - i.e. inside all the accelerated loops
- **Written from perspective of a single thread**
  - Compiler pools statements together for threadblock
  - Limited resource: use sparingly and only specify what's needed
  - Any non-loop variables should be compile-time parameters (CCE)

cache clause examples

- **Example 1:**
  - loop-based stencil
  - inner loop sequential
  - RADIUS should be known at compile time (parameter or cpp)

```
!$acc parallel loop copyin(c)
  DO i = 1,N
   result = 0
!$acc cache(in(i-RADIUS,i+RADIUS),c)
!$acc loop seq
   DO j = -RADIUS,RADIUS
    result = result + c(j)*in(i+j)
   ENDDO
   out(i) = result
  ENDDO
```

cache clause examples

- **Example 2**
  - from "man openacc.examples"
  - multidimensional loopnest
    - stencil only in i,j directions
  - same principle, but...
    - you need to tile the loopnest
    - two options currently:
      - do it explicitly
        - DO jb = 1,N,JBS
        - DO j = jb,MIN(jb+JBS-1,N)
        - and similarly for i
      - use CCE directives, as right
    - OpenACC v2.0 will ease this:
      - tile clause for loop directive
      - more on this later in course

```
!$acc loop gang
DO k = 1,N
!dir$ blockable( i, j )
!$acc loop worker
!dir$ blockingsize ( 16 )
  DO j = 1,N
!$acc loop vector
!dir$ blockingsize ( 64 )
    DO i = 1,N
!$acc cache( A(i,j,k), &
!$acc        B(i-1:i+1,j-1:j+1,k) )

      A(i,j,k) = B(i,  j,  k) - &
                 ( B(i-1,j-1,k) &
                 + B(i-1,j+1,k) &
                 + B(i+1,j-1,k) &
                 + B(i+1,j+1,k) ) / 5
    ENDDO
  ENDDO
ENDDO
!$acc end parallel
```

Tuning code performance

- **Remember the Golden Rules of performance tuning:**
  - always profile the code yourself
    - always verify claims like "this is always the slow routine";
    - codes/computers change
  - optimise the real problem running on the production system
    - a small testcase running on a laptop will have a very different profile
  - optimise the right parts of the code
    - the bits that take the most time
    - even if these are not the exciting bits of the code
    - e.g. it might not be GPU compute; it might be comms (MPI), I/O...
  - keep on profiling
    - the balance of CPU/GPU/comms/IO will change as you go
    - refocus your efforts appropriately
  - Keep on checking for correctness
  - Know when to stop (and when to start again)

Tuning OpenACC performance

- **Tuning needs input:**
  - There are three main sources of information; make sure you use them:
    - Compiler feedback (static analysis)
      - loopmark files (-hlist=a) for CCE; -Minfo=accel for PGI
    - Runtime commentary (CCE only: CRAY_ACC_DEBUG=1 or 2 or 3)
    - Code profiling
      - CrayPAT
      - Nvidia compute profiler
      - pgprof for PGI

Tuning OpenACC codes

- **The main optimisation is minimising data movements**
- **How can I tell if data locality is important?**
  - CrayPAT will show the proportion of time spent in data transfers
    - May need to compile CCE with -hacc_model=auto_async_none to see this
  - Loopmark comments will tell you which arrays might be transferred
    - Compile CCE with -hlist=a and look at .lst files
  - Runtime commentary will tell you which arrays actually moved
    - and how often and when in the code
    - Compile as usual, export/setenv CRAY_ACC_DEBUG=2 at runtime
      - use the runtime API to control the amount of information produced

Tuning OpenACC data locality

- **What can I do?**
  - Use data regions to keep data resident on the accelerator
    - Understanding how data flows in application call tree is crucial, but tricky
  - Only transfer the data you need
    - if only need to transfer some of an array (e.g. halo data, debugging values),
    - rather than use copy* clause, use create and explicit update directives
    - packing/sending a buffer may be faster than sending strided array section
  - Overlap data transfers with other, independent activities
    - use async clause on update directive; then wait for completion later
    - typical situations:
      - pipelining; send one chunk while another processes on the GPU
      - task-based overlap; can be hard to arrange
        - typical use case: pack halo buffer and transfer to CPU while GPU updates bulk
  - Beware of GPU memory allocation overheads
    - if a routine using big temporary arrays is called many times, even create clause can have a big overhead
    - maybe keep array(s) allocated between calls (add to higher data region)
      - add it to a higher data region as create and use present clause in subprogram
    - (not good for a memory-bound code, of course)

Kernel optimisation

- **Next optimisation: make sure all the kernels vectorise**
  - How can I tell if this is a problem?
    - if a kernel is surprisingly slow on accelerator
      - in a wildly different place in the the profile compared to running on CPU
    - examine the loopmark compiler commentary files
  - loop iterations should be divided over both the threads in a threadblock (vector) and over the threadblocks (gang)
    - CCE: you should see either:
      - If a single loop is divided over both levels of parallelism, look for: Gg
      - If two different loops divided, look for G and 2 g-s (maybe with numbers between)
  - generally want to vectorise the innermost loop
    - usually fastest-moving array index, for coalescing
  - if not, can the inner loop be vectorised?
    - i.e. can loop iterations be computed in any order?
    - if not, rewrite code
      - avoid loop-carried dependencies
        - e.g. buffer packing: calculate rather than increment
      - these rewrites will probably perform better on CPU also

Replace:

```
i = 0
DO y = 2,N-1
   i = i+1
   buffer(i) = a(2,y)
ENDDO
buffsize = i
```

By:

```
DO y = 2,N-1
   buffer(y-1) = a(2,y)
ENDDO
buffsize = N-2
```

Forcing compiler to vectorise

- **If the loop is vectorisable, guide the compiler**
  - a gentle hint:
    - put "acc loop independent" directive above this loop
    - could also use CCE directive "!dir$ concurrent"
      - see "man intro_directives" for details
  - a direct order:
    - put "acc loop vector" directive above this loop
  - check the code is still correct and running faster, though:
    - the compiler might not be vectorising for a good reason

- **If the inner loop is vectorising but performance is still bad**
  - is the inner loop really the one to vectorise in this case?
    - in this example, we should vectorise the i-loop
      - because we happen to know mmax is small here
  - put "acc loop seq" directive above m-loop
    - then executed redundantly by every thread
    - also t is now an i-loop private scalar
      - rather than a reduction variable (which is slower)
  - probably also want to reorder array c for speed
    - c(i,m) gives much coalesced memory accesses
    - want vector index to be fastest-moving index

```
!$acc parallel loop
DO i = 1,N
   t = 0
!$acc loop seq
   DO m = 1,mmax
      t = t + c(m,i)
   ENDDO
   a(i) = t
ENDDO
!$acc end parallel loop
```

It's all vectorising, but still performing badly

- **Profile the code and start "whacking moles"**
  - optimise the thing that is taking the time
  - if it really is a GPU compute kernels...
- **GPUs need lots of parallel tasks to work well**
- **First look at loop scheduling using OpenACC clauses**
- **Then might need to consider more extreme measures**
  - source code changes
  - handcoding CUDA kernels

Advanced loop scheduling

- **OpenACC loop schedules are limited by the loop bounds**
  - at least with the current implementation in CCE
  - one loop's iterations are divided over gangs
  - another loop's iterations are divided over threads in a threadblock
- **So...**
  - "tall, skinny" loopnests (j=1:big; i=1:small) won't schedule well
    - if less than 32 iterations won't even fill a warp, so wasted SIMT
  - "short, fat" loopnests (j=1:small; i=1:big) also not good
    - want lots of threadblocks to swap amongst SMs
- **What can we do?**
  - collapse clause is way of increasing flexibility
    - the compiler may use this automatically (look for C in loopmark)
    - no guarantee that it is faster
      - e.g. index rediscovery requires expensive integer divisions
    - need perfectly nested loops for this to work
  - worker clause can also do this

Using the collapse clause

- **Consider a three-level loopnest (i inside j inside k)**
  - needs to be perfectly nested to use collapse
  - Collapse all three loops and schedule across GPU
    - "acc parallel loop collapse(3) gang worker vector" above k-loop
      - probably don't need "gang worker vector" here
  - Schedule inner two loops over threads in threadblock
    - "acc parallel loop gang" above k-loop
    - "acc loop collapse(2) vector" above j-loop
      - don't need "gang"; enough warps are used to cover all the iterations
  - Schedule outer two loops over the threadblocks
    - "acc parallel loop collapse(2) gang" above k-loop
    - "acc loop vector" above i-loop
  - Schedule outer two loops together over entire GPU
    - "acc loop collapse(2) gang worker vector" above k-loop
    - "acc loop seq" above i-loop
  - Schedule k-loop and i-loop together over entire GPU
    - collapsed loops must be perfectly nested; you'll need to reorder the code

workers or vectors?

- **kernel threadblocks are scheduled on SMs**
  - executed as "warps" i.e. vector instructions of length 32
  - threads-per-threadblock>32 automatically decomposed into warps

- **OpenACC makes distinction explicit**
  - worker refers to whole warps (i.e. sets of vector instructions)
    - can be generated explicitly by the user using "!$acc loop worker"
  - vector refers to threads within a warp
    - can be generated automatically by the compiler/runtime
      - vector_length > 32 automatically decomposes into (vector_length/32) workers

- **CCE: only allows one of the above**
  - If you don't specify "!$acc loop worker"
    - vector_length (default 128) automatically partitioned into workers
    - num_workers works the same
  - If you specify "!$acc loop worker"
    - default, or vector_length explicitly set
      - num_workers implicitly set to (vector_length/32)
      - vector_length implicitly set to 32 (see loopmark for information)
    - num_workers explicitly set
      - vector_length set to 32
    - num_workers and vector_length>32 explicitly set
      - Compiler warning that vector_length value is being overridden and set to 32

Scheduling with and without the worker clause

- **The default scheduling**
  - k-loop iterations divided over threadblocks
  - i-loop iterations divided within a threadblock
    - round-robin distribution
      - first thread does i=1, V+1, 2*V+1, ...
      - V is vector_length value (default 128 with CCE)
    - threads automatically grouped into warps
      - first warp does i=1:32, V+1:V+32, ...
  - each thread does all the j-loop iterations

```
!$acc parallel
!$acc loop gang
DO k = 1,N
!$acc loop seq
   DO j = 1,N
!$acc loop vector
      DO i = 1,N
```

- **With explicit loop worker directive**
  - k-loop divided as before
  - i-loop iterations are divided within a warp
    - first thread does i=1, 33, 65, ...
    - each warp does *all* values: i=1:32, 33:64, ...
  - j-loop iterations divided over warps
    - number of warps, W (see previous):
      - either: num_workers value
      - or: vector_length value divided by 32
    - round-robin distribution
      - first warp does j=1, W+1, 2*W+1, ...

```
!$acc parallel
!$acc loop gang
DO k = 1,N
!$acc loop worker
   DO j = 1,N
!$acc loop vector
      DO i = 1,N
```

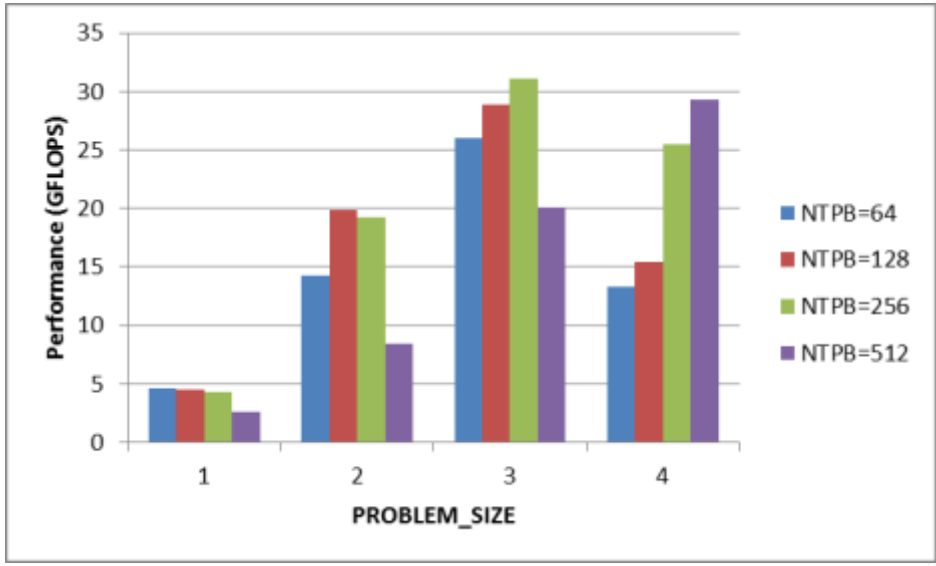
workers or vectors (contd)?

- **So when might we use "!$acc loop worker"?**
- **Perfectly nested loops with one or more low tripcounts**
  - probably better to use the collapse clause
    - e.g. "!$acc loop collapse(2) vector"
    - we'll see this for scalar Himeno shortly
- **Imperfectly nested loops with one or more low tripcounts**
  - may benefit to put "!$acc loop worker" on the middle loop
    - collapse won't work here

Scalar Himeno performance

- **PROBLEM_SIZE=1,2,3,4**
  - For a parallel problem, this simulates strong scaling choices
  - Here we look at all 4 options (all double precision)
- **vector_length choices: NTPB=64,128,256,512**
  - Easy tuning choice, but needs a recompile with CCE
    - but you need to put in vector_length(NTPB) clauses
    - and compile with -DNTPB=<value>
  - CCE defaults to 128

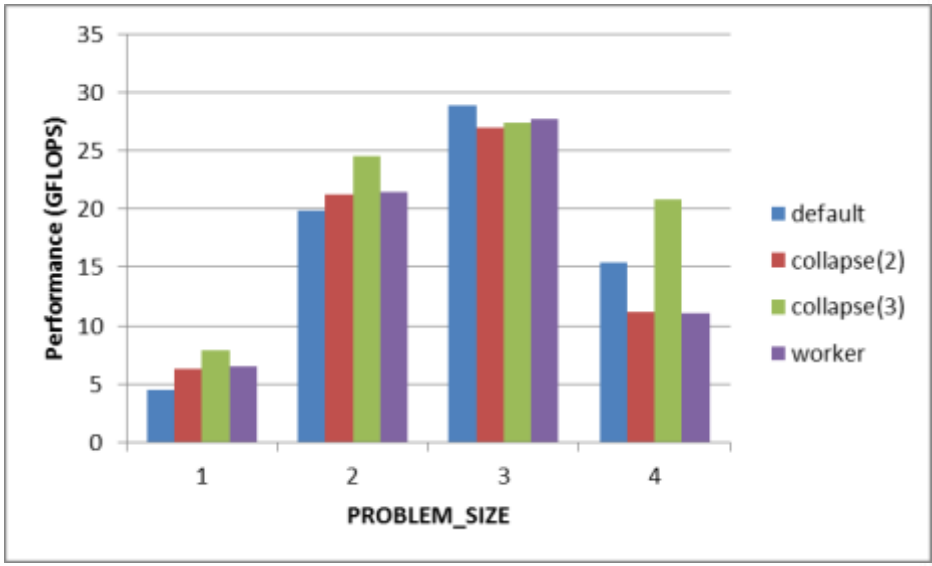
- **This has a BIG effect**
  - best NTPB relative to default:
  - PROBLEM_SIZE=1: + 2%
  - PROBLEM_SIZE=2: default
  - PROBLEM_SIZE=3: + 8%
  - PROBLEM_SIZE=4: +90%



Scalar Himeno performance

- **PROBLEM_SIZE=1,2,3,4**
- **4 algorithm choices for jacobi stencil kernel**
  - default scheduling
  - collapse(2) inner i,j-loops
  - collapse(3) all loops
  - worker schedule for j-loop
- **vector_length: default**

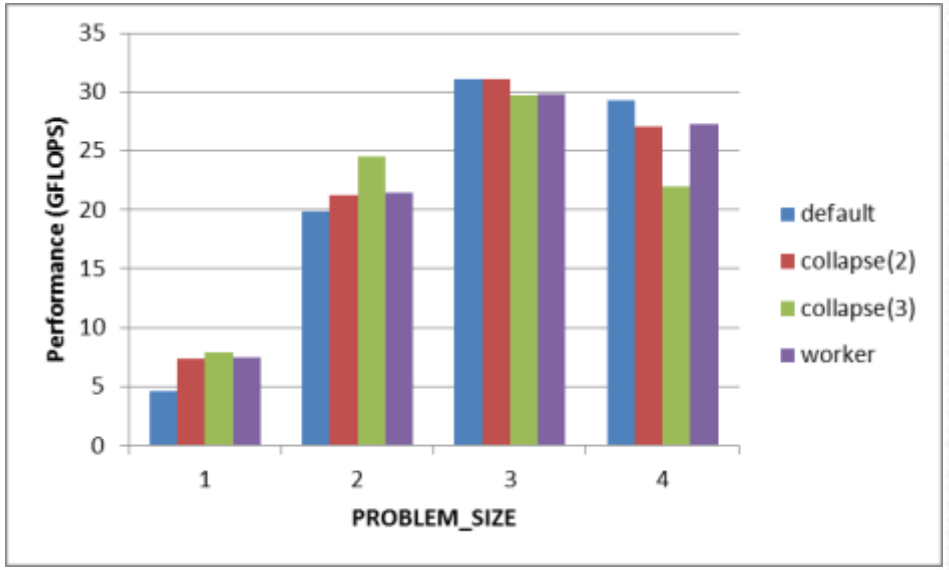
- **Effect also quite big**
  - best NTPB relative to default:
  - PROBLEM_SIZE=1: +76%
  - PROBLEM_SIZE=2: +23%
  - PROBLEM_SIZE=3: default
  - PROBLEM_SIZE=4: +35%



Scalar Himeno performance

- **PROBLEM_SIZE=1,2,3,4**
- **4 algorithm choices for jacobi stencil kernel**
  - default scheduling
  - collapse(2) inner i,j-loops
  - collapse(3) all loops
  - worker schedule for j-loop
- **vector_length: best for each**

- **Final improvements**
  - compared to default
    - scheduling and vector_length:
  - PROBLEM_SIZE=1: +76%
  - PROBLEM_SIZE=2: +23%
  - PROBLEM_SIZE=3: + 8%
  - PROBLEM_SIZE=4: +90%



Scalar Himeno tuning conclusions

- **Tuning can have a big effect, relative to default**
  - It is worth doing, but only after you optimise the data locality
    - data region gave 44x speedup; kernel tuning gave less than 2x
  - We gained something at every problem size (don't reject a mere 2x!)
  - Only explored vector_length and basic scheduling (collapse, worker)
  - The only change was OpenACC directives; CPU version same

- **General conclusions for scalar Himeno:**
  - Larger vector_length suited larger problem sizes
    - 64 best for PS1; 128 best for PS2; 256 best for PS3; 512 best for PS4
  - Once we had optimised vector_length
    - collapse(3) was best for small problems (PS1, PS2)
    - default scheduling was best for large problems (PS3, PS4)

- **What else could we try?**
  - cache clause: could have benefits for the expensive stencil kernel
  - async clause: scalar Himeno is too simple for task-based overlap
    - async is important in the parallel case

Scalar Himeno tuning conclusions

- **Best choice tuning choices (parameters/algorithms)**
  - hard to predict and depends on problem size
    - GPU occupancy (e.g. from COMPUTE_PROFILE=1) guides (but not perfect)
    - you should do the tuning for the production problem, not a test case
    - you should redo the problem for each change in local problem size
      - e.g. when strong scaling a problem
  - you might benefit from using an autotuning framework

- **General conclusions for scalar Himeno:**
  - Larger vector_length suited larger problem sizes
    - 64 best for PS1; 128 best for PS2; 256 best for PS3; 512 best for PS4
  - Once we had optimised vector_length
    - collapse(3) was best for small problems (PS1, PS2)
    - default scheduling was best for large problems (PS3, PS4)

Extreme tuning

- **You've tried tuning with OpenACC clauses**
  - but you think kernel performance can still be improved
  - (and this kernel is the performance-limiter in your application)
- **Now (and only now) you may need... extreme tuning**
- **Some examples:**
  - main source code changes
    - What changes will work?
    - There is no definitive guide
    - Following slides give two cases
  - mixed languages
    - You could handtune the slow kernel in CUDA
    - OpenACC allows interoperability with CUDA (i.e. sharing data)
    - Following slides give a very simple example

Avoiding temporary arrays

- **Perfect loop nests often perform better than imperfect**
  - Imperfect loopnests often use temporary arrays
    - e.g. in a stencil like MultiGrid, to avoid additional duplicated computation
  - With OpenACC, these arrays are privatised; too big for shared memory
    - Imperfect loop nest also means scheduling decisions are restricted
- **Try two approaches; which (if any) faster depends on code**
  - Remove temporary arrays by manually inlining (eliminate array b)
    - one perfect loop nest; cache clause can use shared mem/regs where needed
  - Manually privatise arrays and fission the loopnest (b(i)→b(i,j))

```
DO j = 1,N
 DO i = 0,M+1
  b(i) = a(i,j+1) + a(i,j-1)
 ENDDO
 DO i = 1,M
  c(i,j) = b(i+1) + b(i-1)
 ENDDO
ENDDO
```

```
DO j = 1,N
 DO i = 1,M
  c(i,j) = a(i+1,j+1) + a(i+1,j-1) &
         + a(i-1,j+1) + a(i-1,j-1)
 ENDDO
ENDDO
```

```
DO j = 1,N
 DO i = 0,M+1
  b(i,j) = a(i,j+1) + a(i,j-1)
 ENDDO
ENDDO
DO j = 1,N
 DO i = 1,M
  c(i,j) = b(i+1,j) + b(i-1,j)
 ENDDO
ENDDO
```

More drastic performance optimisations

- **Would reordering your data structures help?**
- **For instance:**
  - Nmax particles each have Smax internal properties
    - code separately combines the internal properties together for each particle
  - CPU code usually stores data as f(Smax,Nmax) or f[Nmax][Smax]
    - good cache reuse when we access all the properties of a particle
  - GPU code would normally parallelise over the particles
    - each thread processes the internal properties of a single particle
    - first warp would attempt vector load of s^{th} prop. of first 32 particles: f(s,1:32)
    - no coalescing (vector load needs contiguous block of memory)
    - very poor performance (even if Smax is small)
  - Better to reorder data so site index fastest: fgpu(Nmax,Smax)
    - vector load of fgpu(1:32,s) now stride-1 in memory
    - if code memory-bandwidth-bound, you will see a big speed-up

- **Quite an effort to reorder data structures in the code**
  - but... may also see benefits on CPU
    - especially with AVX (and longer vectors in future CPU processors)

host_data directive

- **OpenACC runtime manages GPU memory implicitly**
  - user does not need to worry about memory allocation/free-ing
- **Sometimes it can be useful to know where data is held in device memory, e.g.:**
  - so a hand-optimised CUDA kernel can be used to process data already held on the device
  - so a third-party GPU library can be used to process data already held on the device (Cray libsci_acc, cuBLAS, cuFFT etc.)
  - so optimised communication libraries can be used to streamline data transfer from one GPU to another
- **host_data directive provides mechanism for this**
  - nested inside OpenACC data region
  - subprogram calls within host_data region then pass pointer in device memory rather than in host memory

Interoperability with CUDA

```
PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$acc data copy(a)
! <Populate a(:) on device
!  as before>
!$acc host_data use_device(a)
  CALL dbl_cuda(a)
!$acc end host_data
!$acc end data
  <stuff>
END PROGRAM main
```

```
__global__ void dbl_knl(int *c) {
  int i = \
       blockIdx.x*blockDim.x+threadIdx.x;
  if (i < N) c[i] *= 2;
}

extern "C" void dbl_cuda_(int *b_d) {
  cudaThreadSynchronize();
  dbl_knl<<<NBLOCKS,BSIZE>>>(b_d);
  cudaThreadSynchronize();
}
```

- **host_data region exposes accelerator memory address on host**
  - nested inside data region
- **Call CUDA-C wrapper (compiled with nvcc; linked with CCE)**
  - must include cudaThreadSynchronize()
    - Before: so asynchronous accelerator kernels definitely finished
    - After: so CUDA kernel definitely finished before we return to the OpenACC
  - CUDA kernel written as usual
  - Or use same mechanism to call existing CUDA library

More information

- **The standard and quick reference**
- **manual pages**
  - general information: man intro_openacc
    - includes Cray-specific API extensions
  - example programs: man openacc.examples
    - some common usage cases
  - CCE compiler-specific information
    - Fortran: man crayftn
    - C: man craycc
    - C++: man crayCC
    - You'll need module PrgEnv-cray loaded to see these
  - Nvidia Compute Profiler
  - PGI Insider magazine articles
- **support**
  - OpenACC forum
  - Cray email list: openacc-users
  - us!

What's next?

- **OpenACC v1.0 released in November 2011**
  - Cray, PGI and CAPS now offer full support
- **OpenACC v2.0 now being finalised**
  - proposed additions document on OpenACC website
  - discussions and decisions at February's face-to-face meeting
  - expect announcement in next few months
- **We are always interested to hear from you:**
  - bugs (functionality and performance)
  - feature requests
- **Further questions after the course ends?**
  - Please feel free to email me: ahart@cray.com