Case study: the parallel Himeno code

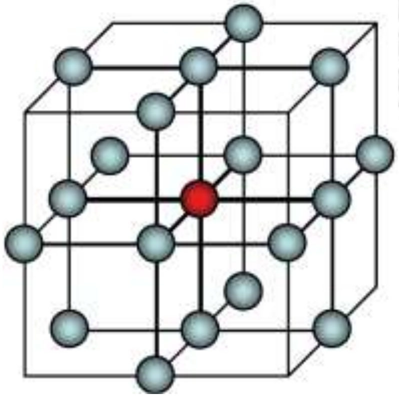
Alistair Hart

Cray Exascale Research Initiative Europe

Contents

- **A parallel code is a scalar code with data transfers**
  - We have looked at how to port a scalar code
  - Here we look at the parallel version of the same code
- **The new feature is the data transfer between processors**
  - which means local data transfers between CPU and GPU
- **This talk looks at the extra things we need to consider**
  - First at a conceptual level
    - the OpenACC part
  - Then some specific points for two different comm models
    - MPI
    - Fortran coarrays

The Himeno Benchmark



- **Parallel 3D Poisson equation solver**
  - Iterative loop evaluating 19-point stencil
  - Memory intensive, memory bandwidth bound
- **Fortran, C, MPI and OpenMP implementations available from http://accc.riken.jp/2444.htm**
- **Fortran Coarray (CAF) version developed**
  - ~600 lines of Fortran
  - Fully ported to accelerator using 27 directive pairs
- **Strong scaling benchmark**
  - XL configuration: 1024 x 512 x 512 global volume
  - Expect halo exchanges to become significant as we scale
  - Async. GPU data transfers and kernel launches to help avoid this

Overall program structure

- Like scalar case:
  - initmt() initialises data
  - jacobi(nn,gosa)
    - does nn iterations
    - stencil update to data
  - called twice:
    - once for calibration
    - once for measurement
- differences:
  - initcomm() routine
    - sets up processor grid

```
PROGRAM himeno
   CALL initcomm   ! Set up processor grid
   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 ',nn,' times'
   PRINT *,' Gosa :',gosa
   PRINT *,' MFLOPS:',xmflops2,'  time(s):',cpu
END PROGRAM himeno
```

The distributed jacobi routine

- iteration loop:
  - fixed tripcount
- jacobi kernel:
  - new pressure array wrk2
  - local residual wgosa
- halo exchange
  - between neighbours
  - uses send, receive buffers
- Allreduce
  - global residual gosa
  - wgosa summed over PEs
- second kernel:
  - p array updated from wrk2

```
DO loop = 1, nn

  compute Jacobi: wrk2, wgosa

  pack halos from wrk2 into send bufs

  exchange halos with neighbour PEs

  Allreduce to sum wgosa across Pes

  copy back wrk2 into p

  unpack halos into p from recv bufs

ENDDO
```

The Jacobi computational kernel (serial)

- The stencil is applied to pressure array p
- Updated pressure values are saved to temporary array wrk2
- Local control value wgosa is computed

```
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)

   s = (s0*a(I,J,K,4)-p(I,J,K))*bnd(I,J,K)
   wgosa = wgosa + ss*ss
   wrk2(I,J,K) = p(I,J,K) + omega * ss
  ENDDO
 ENDDO
ENDDO
```

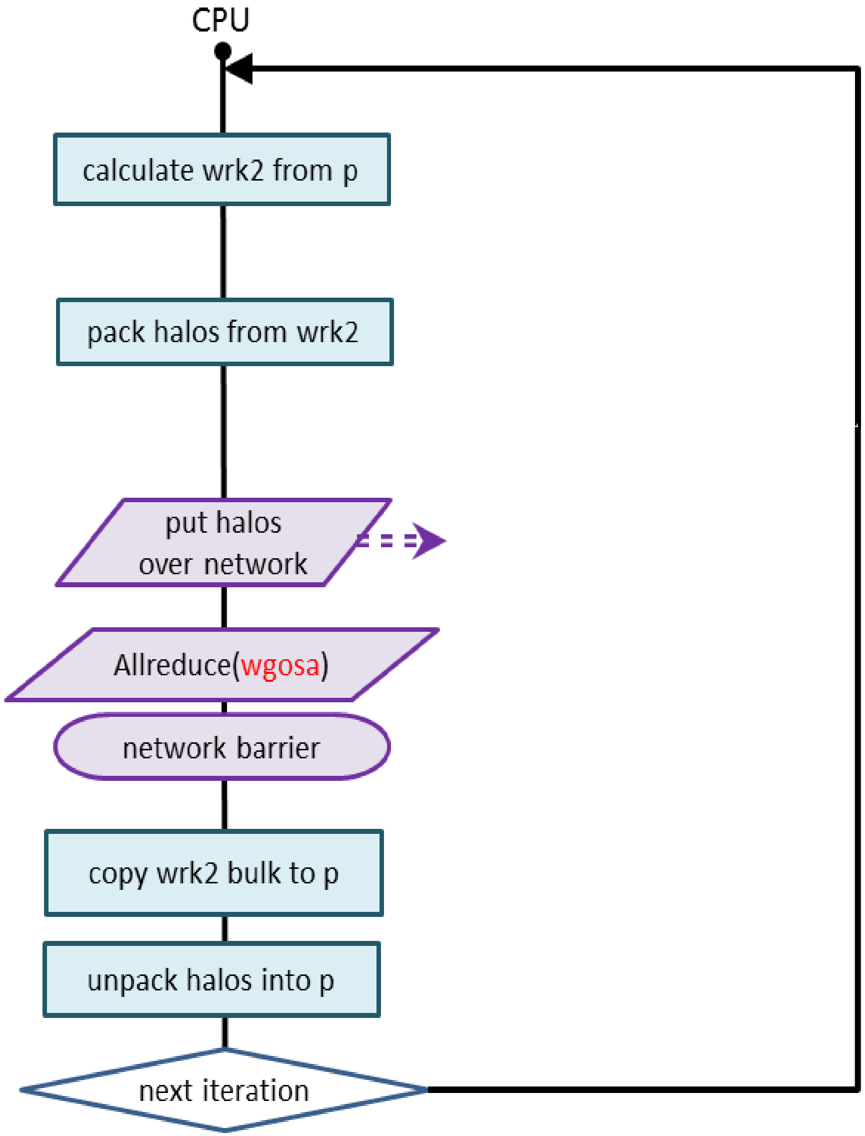
fwd n.n.

n.n.n.

bwd n.n.

Distributed jacobi routine as a flowchart

- **tasks reorganised**
  - take advantage of overlap
    - just network at this point
  - still some freedom:
    - barrier can move relative to
      - Allreduce
      - wrk2→bulk
  - Which is best order?
    - depends on:
      - hardware
      - comms library
      - kernel details
      - local problem size

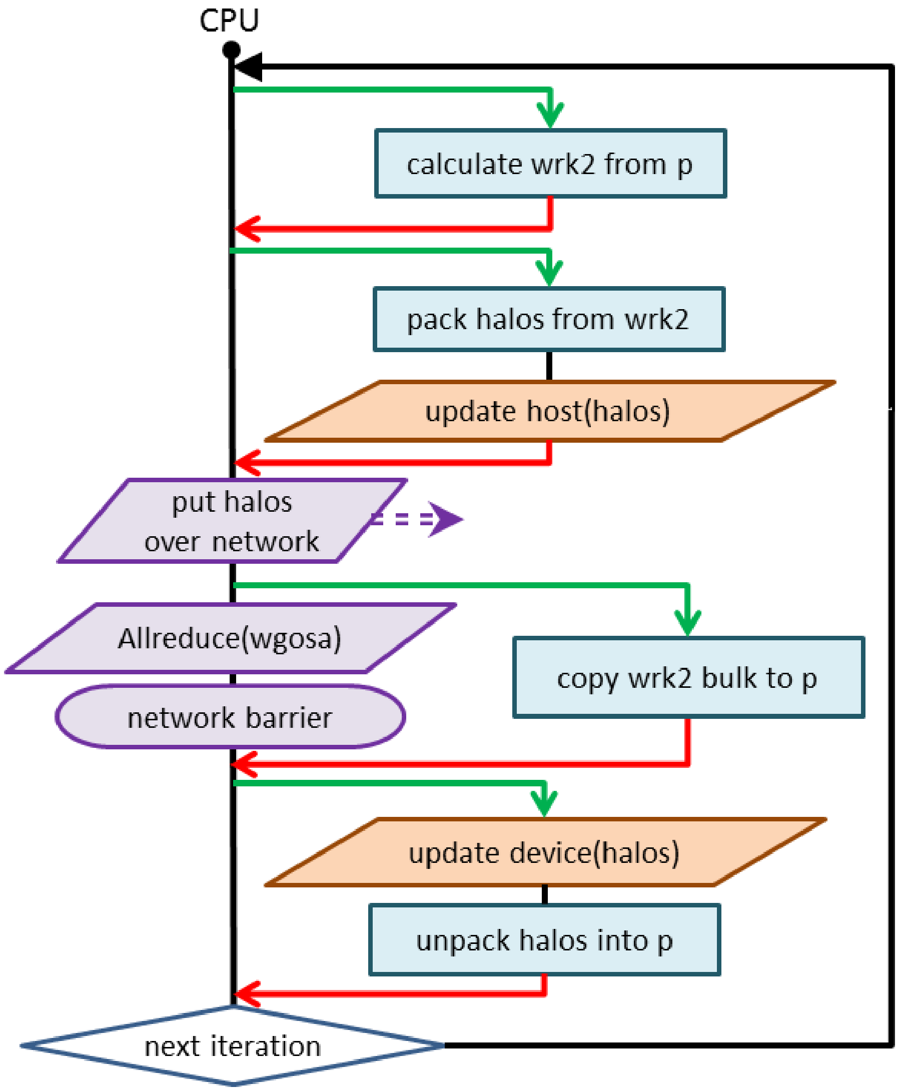


First OpenACC port

- **Loopnest kernels**
  - stencil
  - pack halos
  - wrk2→bulk
  - unpack halos
- **acc updates**

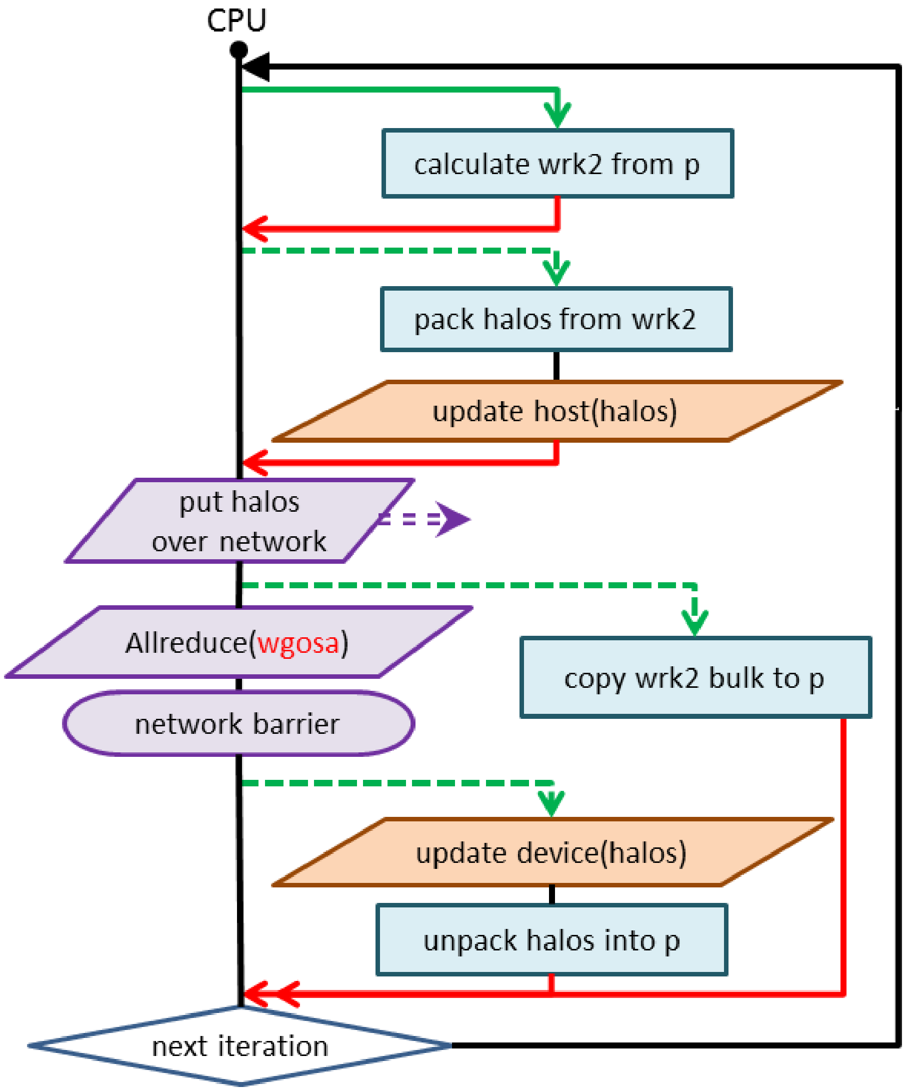
- **Some additional overlap**
  - GPU kernels asynchronous

  - wrk2→bulk
    - can overlap comms



Asynchronicity

- **async handle**
  - allows task overlap on GPU
- **Dotted lines are async**
  - halo pack
    - can overlap x, y, z directions
  - halo unpack
    - can overlap x, y, z directions
    - also overlap with wrk2→bulk
- **Red lines are sync points**



Packing and transferring send buffers

- **Use async clause**
  - separate handles for overlap
  - one per direction
    - three in total
    - integers, e.g. 1, 2, 3
  - what about six?
    - one each for up and down
    - not as efficient
      - but this is a tuning option
  - global wait
    - all 3 streams completed

```
!$acc parallel loop async(xstrm)
DO k = 2,kmax-1
  DO j = 2,jmax-1
    sendbuffx_dn(j,k)=wrk2(2,j,k)
    sendbuffx_up(j,k)=wrk2(imax-1,j,k)
  ENDDO
ENDDO
!$acc end parallel loop

!$acc update host &
!$acc   (sendbuffx_dn,sendbuffx_up)  &
!$acc   async(xstrm)

! same for y with async(ystrm)
! same for z with async(zstrm)

!$acc wait
<send the 6 buffers>
```

Packing and transferring send buffers (better)

- **The biggest bottleneck**
  - PCIe link serialises:
    - one buffer at a time
  - So:
    - as soon as one direction done
    - send these buffers over network
  - Ordering:
    - Can we guarantee x ready first?
    - No, but likely
  - OpenACC does not have waitany directive
    - so go with most likely ordering

```
!$acc parallel loop async(xstrm)
<pack the two x buffers>
!$acc end parallel loop

!$acc update host &
!$acc  (sendbuffx_dn,sendbuffx_up) &
!$acc  async(xstrm)

! same for y with async(ystrm)
! same for z with async(zstrm)

!$acc wait(xstrm)
<send the two x buffers>

!$acc wait(ystrm)
<send the two y buffers>

!$acc wait(zstrm)
<send the two z buffers>
```

Transferring and unpacking recv buffers

- **Copying wrk2 into p**
  - very quick
  - but can overlap network
- **When messages complete**
  - copy and unpack recv buffers
  - three parallel streams
- **wait**
  - ensures all 4 streams complete

```
!$acc parallel loop async(bstrm)
<wrk2 -> p>
!$acc end parallel loop

<network barrier>

!$acc update device &
!$acc  (recvbuffx_dn,recvbuffx_up) &
!$acc   async(xstrm)

!$acc parallel loop async(xstrm)
<unpack the two x buffers>
!$acc end parallel loop

!  same for y with async(ystrm)
!  same for z with async(zstrm)

!$acc wait

<end iteration loop>
```

Data region considerations

- **Twelve extra buffers**
  - 3 physical dimensions: x, y, z
  - 2 directions of transfer: up, down
  - Separate buffers for send and receive

- **These buffers need to be added to the data region(s)**
  - if just in jacobi() data region
    - they should be create
      - allocated each time jacobi() is called, no data transfer (except explicit updates)

  - or in outer data region
    - create in main data region
    - present in jacobi() data region

  - if jacobi() routine was called a lot, better to allocate once in parent

Communications models

- **Two domain decompositions strategies**
  - Both message passing:
    - MPI
    - Fortran coarrays (CAF)

- **A lot of similarities**
  - a few subtle differences

MPI

- **Use nonblocking MPI calls**
  - MPI_IRECV
    - post these receives early; first thing in iteration loop
  - MPI_ISEND
    - call these in 3 sets (x, y, z) of two (up, down)
      - after each of async streams xstrm, ystrm, zstrm separately completes
  - network barrier:
    - MPI_WAITALL(12,send_handles,recv_handles)
    - ensures all buffers arrive (and send buffers can be reused)
  - But, PCIe transfer of recv buffers is a bottleneck
    - better to start transfer of messages to GPU as soon as they arrive
      - don't know which order 6 message will arrive in
    - MPI_WAITANY(6,recv_handles)
      - As each arrives, start async stream of: update, then unpack kernel
    - After all recvs completed:
      - MPI_WAITALL(6, send_handles)
- **Could investigate use of nonblocking collective for gosa**
  - MPI 3.0

Fortran coarrays

- **Introduced in Fortran2008**
  - previously known as Co-array Fortran (CAF) language extension
- **PGAS feature**
  - single-sided communication model

- **Simplifies the code**
  - e.g. to send a buffer:
    - recvbuffx_up(:,:)[myx-1,myy,myz] = sendbuffx_dn(:,:)
  - send buffer is a (local) array, but a (global) co-scalar
  - recv buffer is a (local) array and a (global) coarray

  - no acknowledgement, receipt etc. for message being sent
  - RDMA put: remote "image" doesn't know when data arrives
  - user has to handle all synchronisation points

  - gives great potential for overlapping comms with compute etc.

CAF and Himeno

- **"put" operations are more efficient than "get"**
  - i.e. put remote coarray on the left of the "="
    - recvbuffx_up(:,:)[myx-1,myy,myz] = sendbuffx_dn(:,:)
- **But, Fortran standard says that:**
  - Not only should this guarantee z = x:
    - y = x
    - z = y
  - But also, so should this:
    - y[remote] = x
    - z = y[remote]
  - So compiler forced to add a lot of (usually unnecessary) sync points
  - To avoid these, need CCE directive above CAF statements
    - !$pgas defer_sync

Using CAF for parallel Himeno with OpenACC

- **send buffers**
  - co-scalars, no problem using them in OpenACC directives
- **recv buffers**
  - co-arrays, forbidden from being in OpenACC directives
    - to prevent race conditions
  - so we need to:
    - copy local recv (coarray) buffer into temporary (co-scalar) array on CPU
    - use the temporary array in the update, parallel loop kernels
    - additional CPU buffer copy potentially affects performance/scalability
- **data region**
  - send buffers and temporary buffers (all co-scalars) go in data region
    - using create or present clause

CAF synchronisation

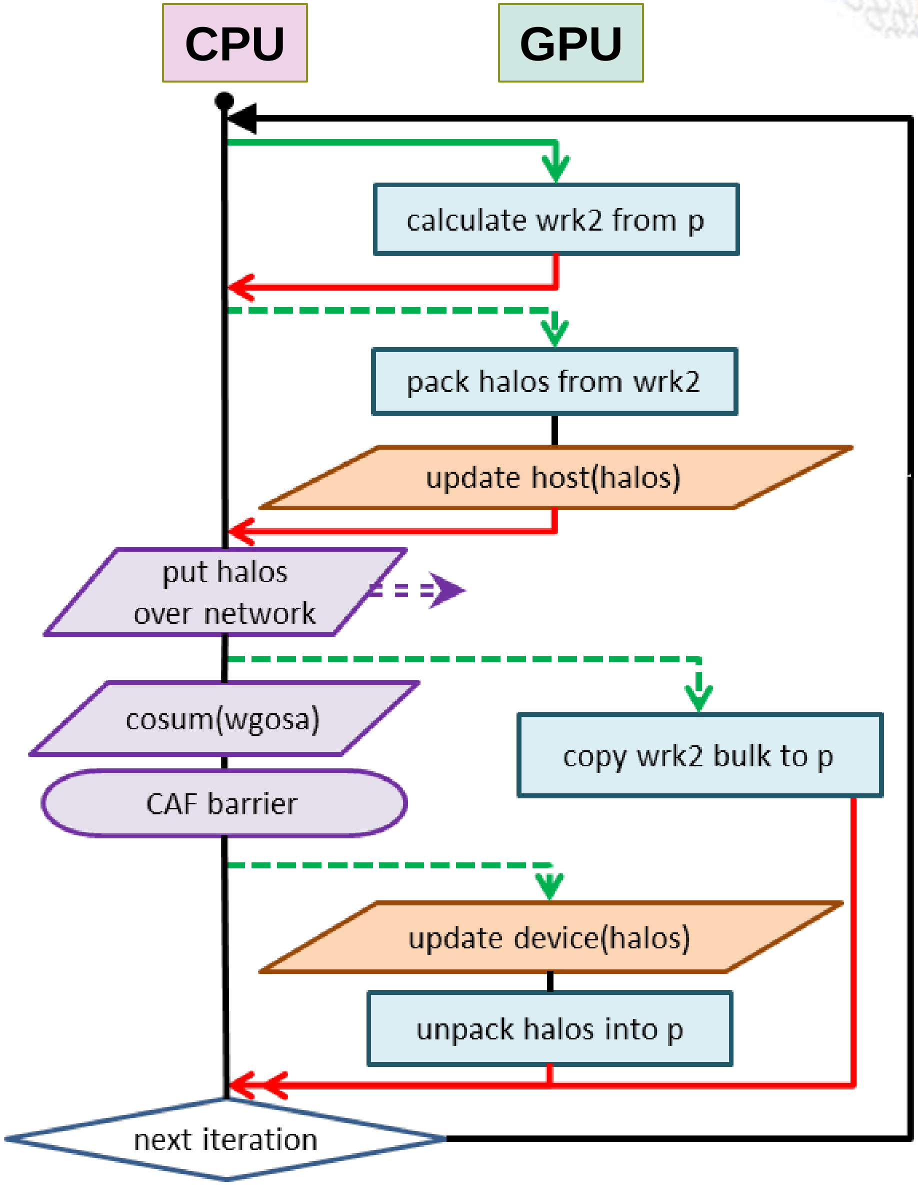
- **Network synchronisation**
  - needed to guarantee the halo buffers have arrived
  - sync all

  - but we are also globally summing the residual
    - scalar wgosa needs to be a co-scalar (as used in OpenACC reduction)
    - copy wgosa to local part of coarray cgosa
    - gosa = cosum(cgosa)

  - cosum partially synchronises the images
    - sync all would now be overkill
    - sync memory instead completes the synchronisation

  - this guarantees all messages have completed
    - no equivalent of MPI_WAITANY to relieve PCIe recv buffer congestion

Final CAF implementation



OpenACC / CAF version

- **Total number of lines in the original Himeno MPI-Fortran code:** 629

- **Total number lines in the modified version with coarrays and accelerator directives:** 554
  - don't need MPI_CART_CREATE and the like

- **Total number of accelerator directives:** 27
  - plus 18 "end" directives

Benchmarking the code

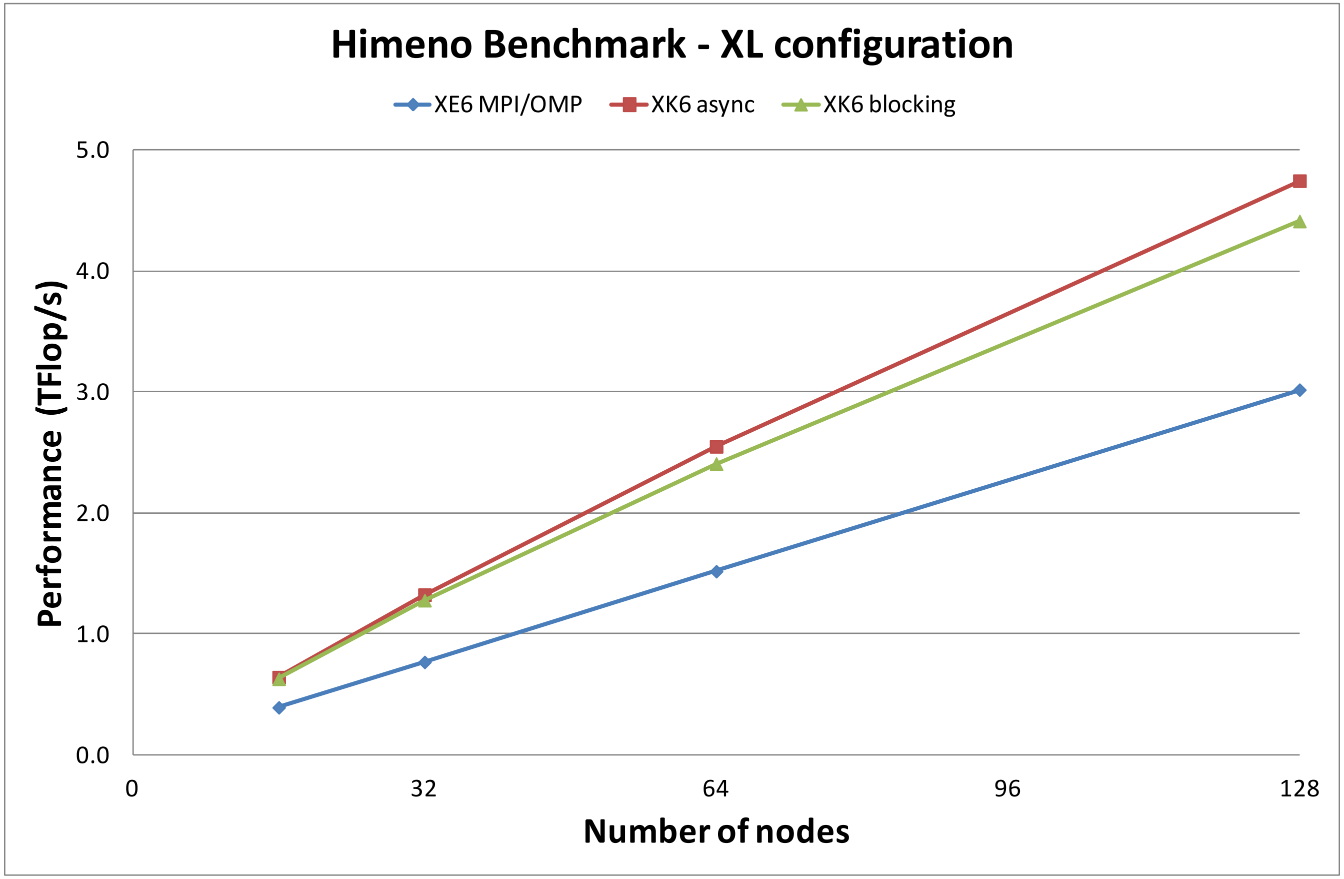
- **Cray XK6 configuration:**
  - Single AMD IL-16 2.1GHz nodes, 16 cores per node
  - Nvidia Tesla X2090 GPU, 1 GPU per node
  - Running with 1 PE (GPU) per node
  - Himeno case XL needs at least 16 XK6 nodes
  - Testing blocking and asynchronous GPU implementations

- **Cray XE6 configuration:**
  - Dual AMD IL-16 2.1 GHz nodes, 32 cores per node
  - Running on fully packed nodes: all cores used
  - Depending on the number of nodes, 1-4 OpenMP threads per PE are used

- **All comparisons are for strong scaling on case XL**

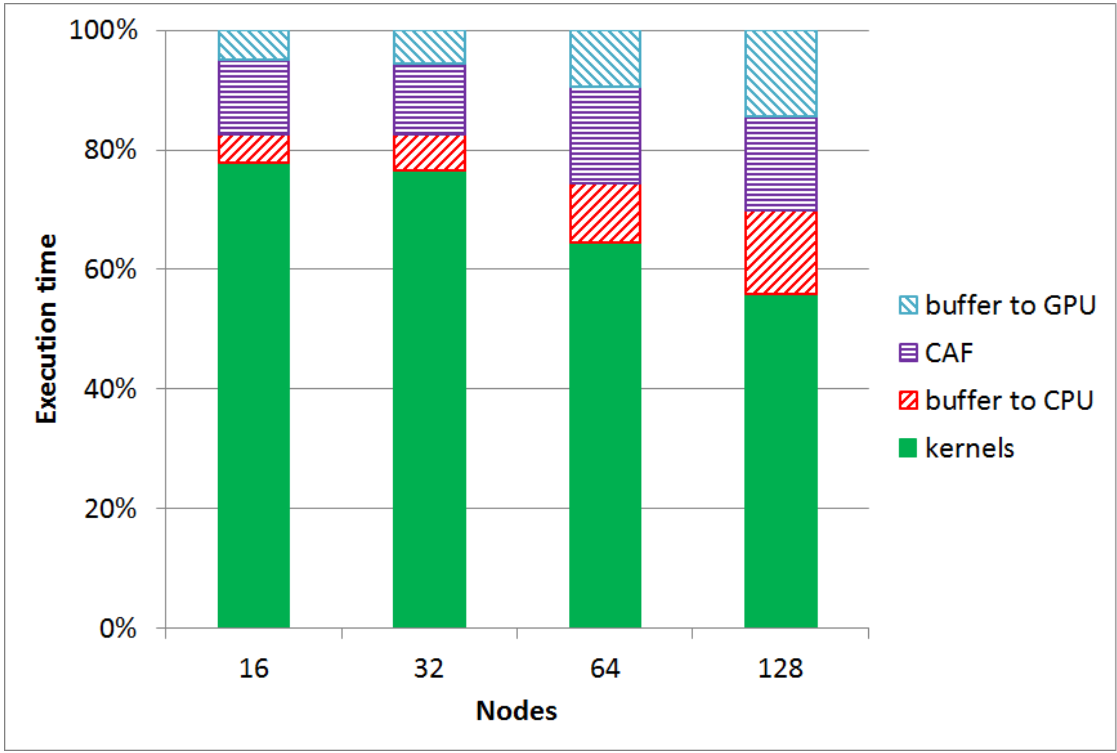
Himeno performance

- XK6 GPU is about 1.6x faster than XE6
- OpenACC async implementation is ~ 8% faster than OpenACC blocking



Himeno code breakdown

- **Host/GPU transfers take more time than the halo exchange (network)**
  - this code would benefit from an efficient direct GPU-GPU communication
- **On 128 nodes, ~55% of the time is spent in the GPU compute kernel**


Execution time
Nodes
buffer to GPU
CAF
buffer to CPU
kernels