A quick tour of OpenACC & Co.

or

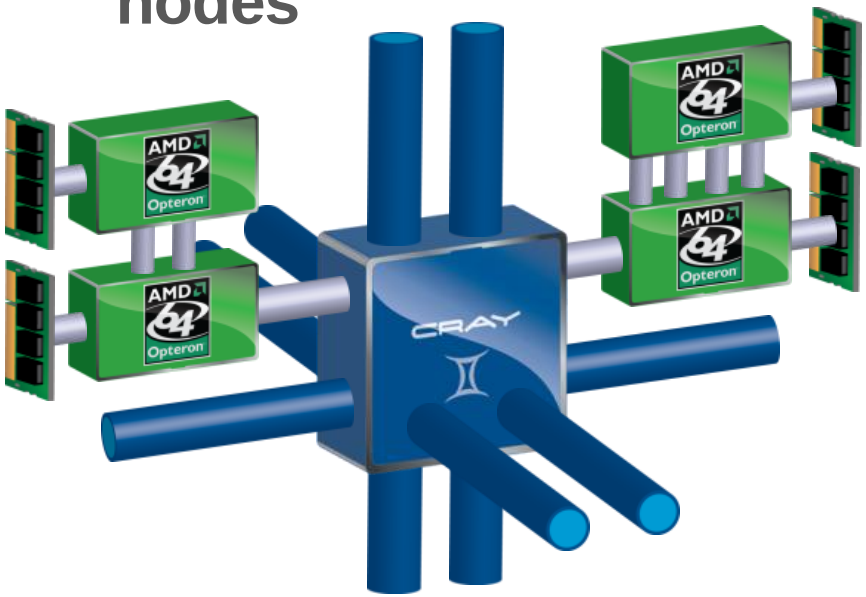
How should I program this accelerator ?

Roberto Ansaloni
Cray Italy
roberto@cray.com

The Cray XK6 heterogeneous node

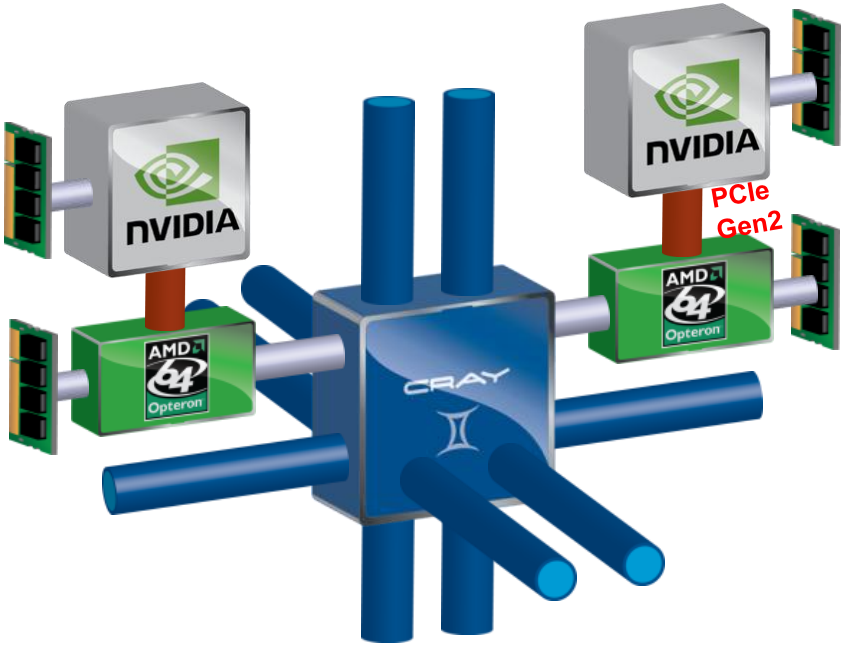
The Cray **XE6** compute node

- Built around the Gemini Interconnect
- Each Gemini ASIC provides 2 NICs enabling it to connect 2 dual-socket nodes

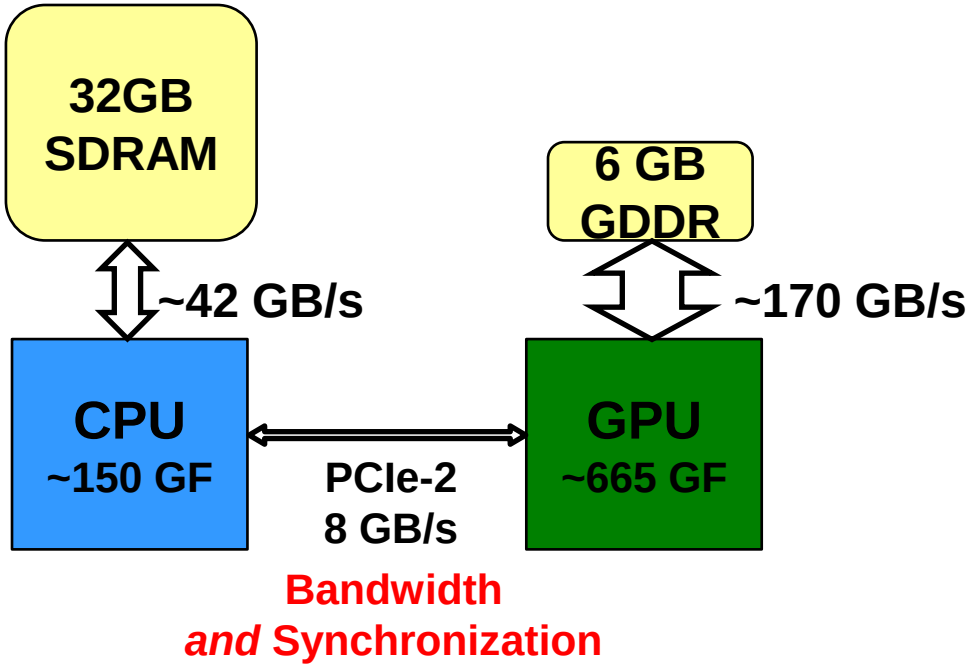


The Cray **XK6** compute node

- Replace one CPU socket with a GPU



Issues with today's accelerators (Fermi)



- **This is a short-lived situation**
  - Solutions coming from several vendors (NVIDIA, AMD,…)
- **Trick is to keep kernel data structures resident in GPU memory as much as possible**
  - Avoids copying between CPU and GPU
  - Use async, non-blocking, communication, multi-level overlapping

Roberto's recipe

- **KISS Principle: Keep It Simple**
  - As simple as possible
  - Unless you really really really need performance today
- **Use a flexible and portable approach**
  - GPU architectures have changed and will change: don't stick to a specific one
  - GPUs are just one kind of accelerators
- **Exploit libraries**
  - Exploit work done by smart people
- **Don't forget Amdahl**
  - Amdahl who ?

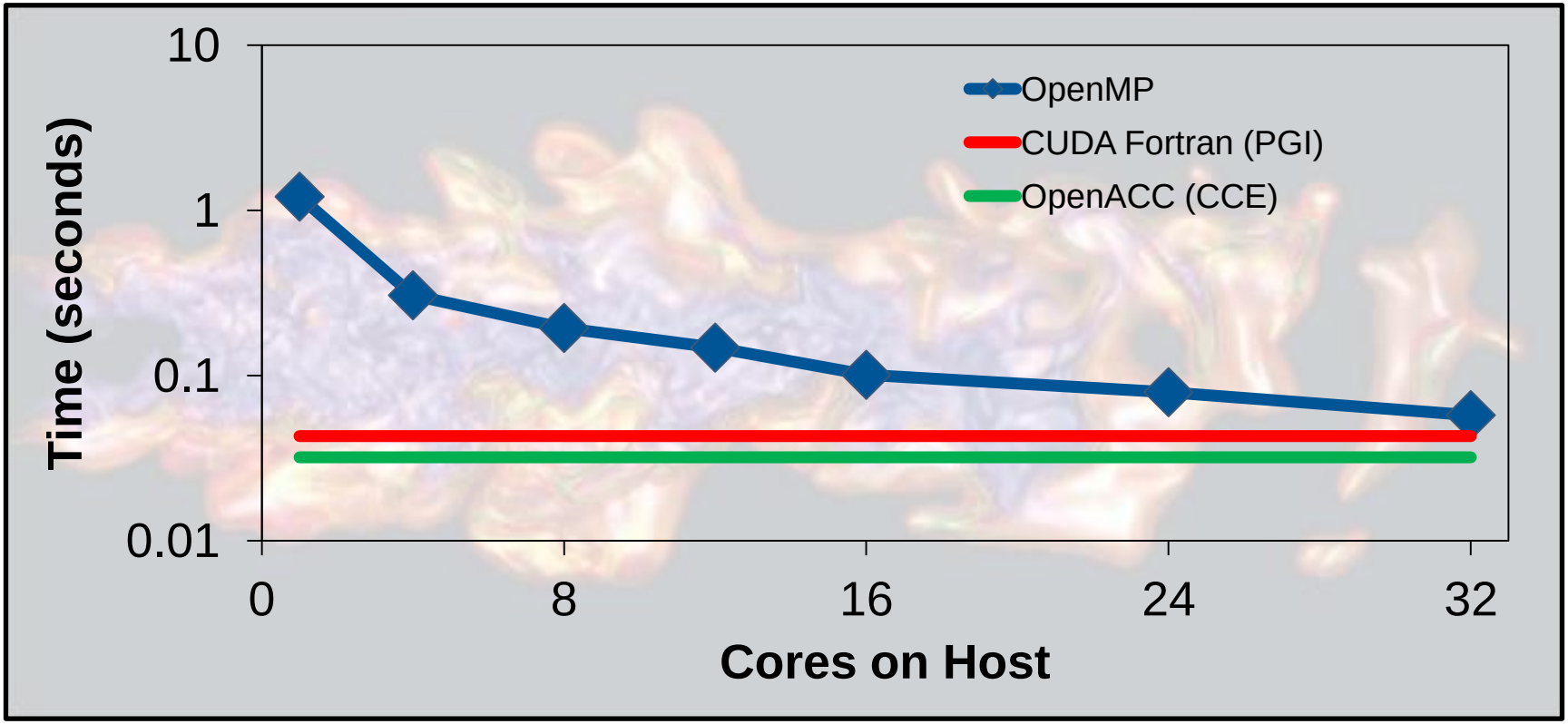
How to program an accelerator ?

- **The hard way: CUDA, OpenCL**
  - All are quite low-level
  - CUDA is closely coupled to the GPU
  - User needs to rewrite kernels in specialist language
  - Hard to write and debug
  - Hard to optimise for specific GPU
  - Hard to port to new accelerator
  - Hard to add new functionality

- **A simpler approach: accelerator directives**
  - Several initial proposals: PGI directives, OpenMP (Cray)
  - Currently merged into OpenACC
  - Based on original source code (e.g. Fortran, C, C++)
  - Easier to maintain/port/extend code
  - Can support future accelerators
  - Possible performance sacrifice

Performance compared to CUDA

- Is there a performance gap relative to explicit low-level programming model? **Typically 10-15%, sometimes none.**
- Is the performance gap acceptable? **Yes.**
  - e.g. S3D comp_heat kernel (ORNL application readiness):



OpenACC
DIRECTIVES FOR ACCELERATORS

CRAY

- **A common directive programming model for today's GPUs**
  - Announced at SC11 conference
  - Offers portability between compilers
    - Drawn up by: NVIDIA, Cray, PGI, CAPS
    - Multiple compilers offer:
      - portability, debugging, permanence
  - Works for Fortran, C, C++
    - Standard available at www.OpenACC-standard.org
    - Initially implementations targeted at NVIDIA GPUs
- **Current version: 1.0 (November 2011)**
- **Compiler support:**
  - Cray CCE: partial now, complete in 2012
  - PGI Accelerator: released product in 2012
  - CAPS: released product in 2012

CAPS CRAY THE SUPERCOMPUTER COMPANY NVIDIA The Portland Group

OpenMP accelerator directives

- **A common programming model for tomorrow's accelerators**
- **An established open standard is the most attractive**
  - portability; multiple compilers for debugging; permanence
- **Subcommittee of OpenMP ARB**
  - includes most major vendors + others (e.g. EPCC)
  - co-chaired by Cray (James Beyer)
  - aiming for OpenMP 4 (2012?)
- **Targets Fortran, C, C++**
- **Current version: draft**
- **Cray compiler provides reference implementation for ARB**
  - Of draft standard at present (CCE 8.0)
  - Will track the standard as it evolves
- **Converting from OpenACC to OpenMP will be straightforward**

OpenACC Execution model

- **Host-directed execution with attached GPU**
- **Main program executes on “host” (i.e. CPU)**
  - Compute intensive regions offloaded to the accelerator device under control of the host.
- **“device” (i.e. GPU) executes parallel regions**
  - typically contain “kernels” (i.e. work-sharing loops), or
  - kernels regions, containing one or more loops which are executed as kernels.
- **Host must orchestrate the execution by:**
  - allocating memory on the accelerator device,
  - initiating data transfer,
  - sending the code to the accelerator,
  - passing arguments to the parallel region,
  - queuing the device code,
  - waiting for completion,
  - transferring results back to the host, and
  - deallocating memory.

A first OpenACC example

Execute a loop nest on the GPU

- Compiler does the work
- Data movement
  - allocates/frees GPU memory at start/end of region
  - moves of data to/from GPU

```
!$acc parallel loop
DO j = 1,M
  DO i = 2,N-1
    c(i,j) = a(i,j) + b(i,j)
  ENDDO
ENDDO
!$acc end parallel loop
```

- Loop schedule: spreading loop iterations over PEs of GPU
- Tune default behaviour with optional clauses on directives

Another example

- Two accelerator parallel regions
- Compiler creates two kernels
  - First kernel initialises array
  - Compiler will determine copyout(a)
- Second kernel updates array
  - Compiler will determine copy(a)
- Breaking parallel region=barrier
- No barrier directive
- The code can still be compiled for CPU

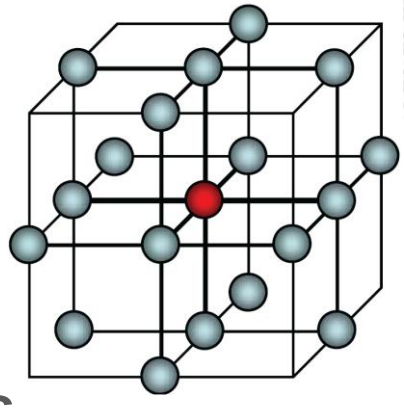
```
PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$acc parallel loop
  DO i = 1,N
   a(i) = i
  ENDDO

!$acc parallel loop
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
<stuff>
END PROGRAM main
```

Let's control data movement

- Now added a data region
  - Specified arrays only moved at boundaries of data region
- No compiler-determined movements for data regions
- Other directives/clauses are available to allow a more direct control of data movements
  - present clause
  - !$acc update [host | device]

```
PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$acc data copyout(a)
!$acc parallel loop
  DO i = 1,N
   a(i) = i
  ENDDO
!$acc parallel loop
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$acc end data
  <stuff>
END PROGRAM main
```

A case study: 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/HPC_e/himenobmt_e.html
- **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
  - Use asynchronous GPU data transfers and kernel launches to help avoid this

The Jacobi computational kernel (serial)

- The stencil is applied to pressure array p
- Updated pressure values are saved to temporary array wrk2
- Control value wgosa is computed
- In the benchmark this kernel is iterated a fixed number of times (nn)

```
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)
   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.

The distributed implementation

- The outer loop is executed fixed number of times
- The Jacobi kernel is executed and new pressure array wrk2 and control value wgosa are computed
- The p array is updated with wrk2 values
- The halo region values are exchanged between neighbor PEs using send and receive buffers
- The maximum wgosa value is computed with an Allreduce operation across all the PEs

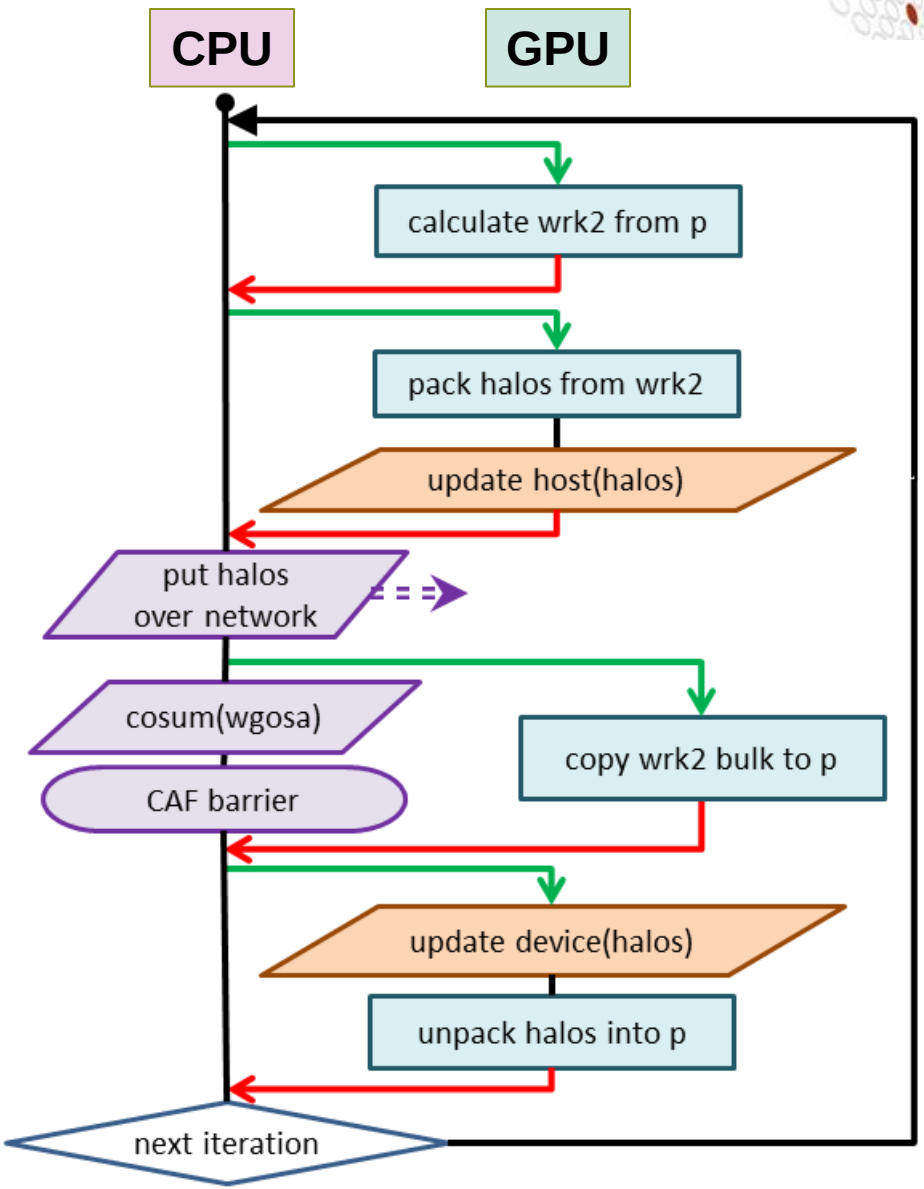
```
DO loop = 1, nn

   compute Jacobi: wrk2, wgosa

   copy back wrk2 into p

   pack halo from p into send buf

   exchange halos with neighbor PEs

   unpack halo into p from recv buf

   Allreduce to sum wgosa across Pes

ENDDO
```

Porting Himeno to the Cray XK6

- **Several versions tested, with communication implemented in MPI or Fortran coarrays**
- **GPU version using OpenACC accelerator directives**
- **Arrays reside permanently on the GPU memory**
- **Data transfers between host and GPU are:**
  - Communication buffers for the halo exchange
  - Control value
- **Cray XK6 timings compared to best Cray XE6 results (hybrid MPI/OpenMP)**

The Himeno GPU code structure

- **GPU performs**
  - Jacobi kernel
  - Halo buffers packing/unpacking
  - Pressure update
- **Host/device communication**
  - Halo region buffers transfer
  - Control value wgosa
- **CAF communication**
  - Remote halo buffers put
  - Global wgosa sum



Jacobi kernel on the GPU

- The GPU kernel for the main loop is created with the `parallel loop` directive
- The scoping of the main variables is specified earlier with the `data` directive - no need to replicate it in here
- wgosa is computed by specifying the `reduction` clause, as in a standard OpenMP parallel loop
- `vector_length` clause is used to indicate the number of threads within a threadblock (compiler default 128)

```
DO loop=1,nn
  gosa = 0
  wgosa = 0
!$acc parallel loop                    &
!$acc&  private(s0,ss)                 &
!$acc&  reduction(+:wgosa)             &
!$acc&  vector_length(256)
  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 )&
        ...
        wgosa = wgosa + SS*SS
      ENDDO
    ENDDO
  ENDDO
```

Coarray implementation

- Coarrays are used to perform the halo exchange
- Non-blocking communication needs `pgas defer_sync` directive
- Programmer now responsible for data synchronization
- By deferring sync point, network communications can be overlapped with CPU or GPU activity
- Updating p from wrk2 (on GPU) overlapped with halo exchange

N.B.
no `sync all`
CAF intrinsic `COSUM` has loose synchronisation (so does need `sync memory` first).

```
!dir$ pgas defer_sync
recvbuffz_up(:,:)[myx,myy,myz-1]= &
   sendbuffz_dn(:,:)
 ...
!$acc parallel loop
DO k = 2,kmax-1
  DO j = 2,jmax-1
    DO i = 2,imax-1
      p(i,j,k) = wrk2(i,j,k)
    ENDDO
  ENDDO
ENDDO
!$acc end parallel loop

sync memory
CO_SUM(wgosa)

!$acc update device &
!$acc& (recvbuffz_dn,recvbuffz_up)
```

OpenACC / CAF version

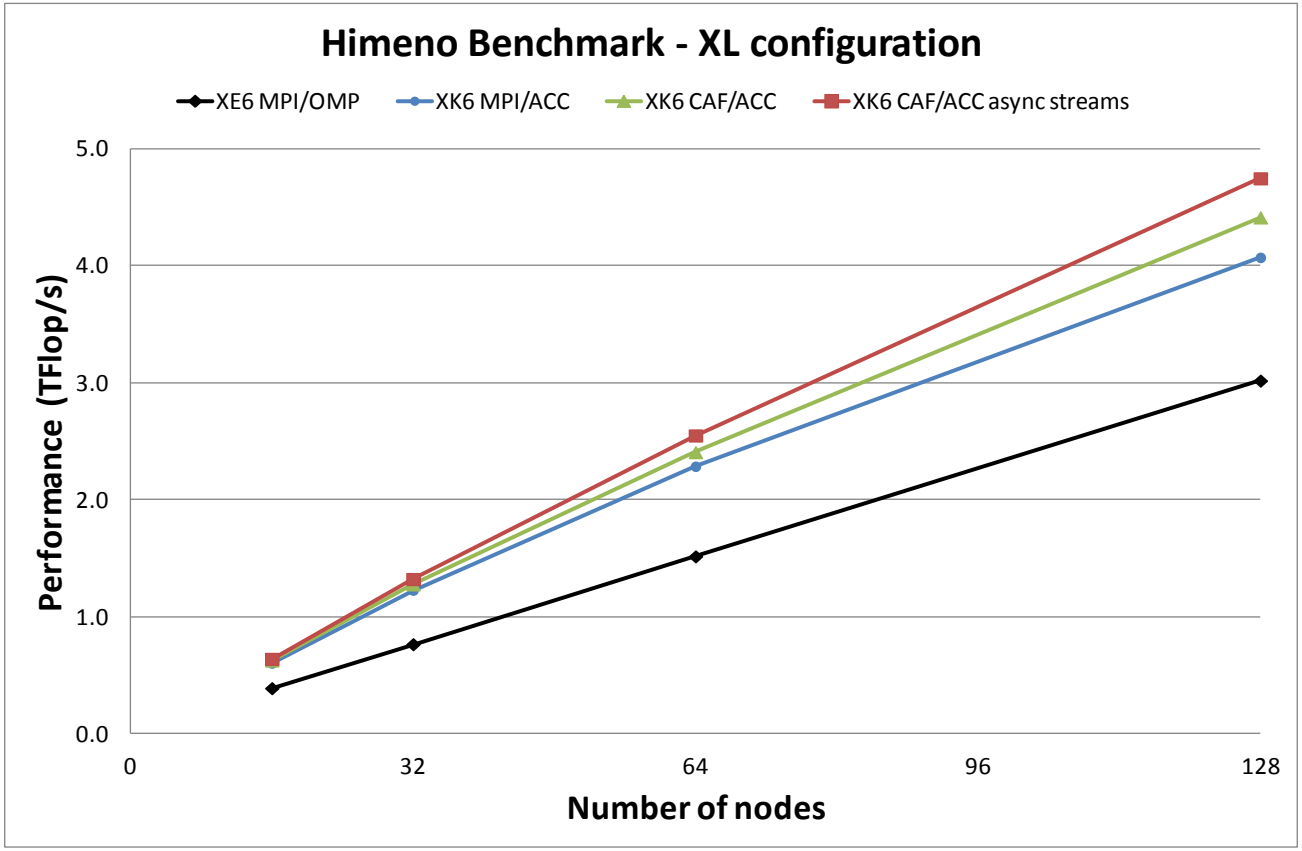
- **Total number of lines in the original Himeno MPI-Fortran code:** 629
- **Total number lines in the modified version with coarrays and accelerator directives:** 554
  - don't need MPI_CART_CREATE and the like
- **Total number of accelerator directives:** 27
  - plus 18 "end" directives

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**

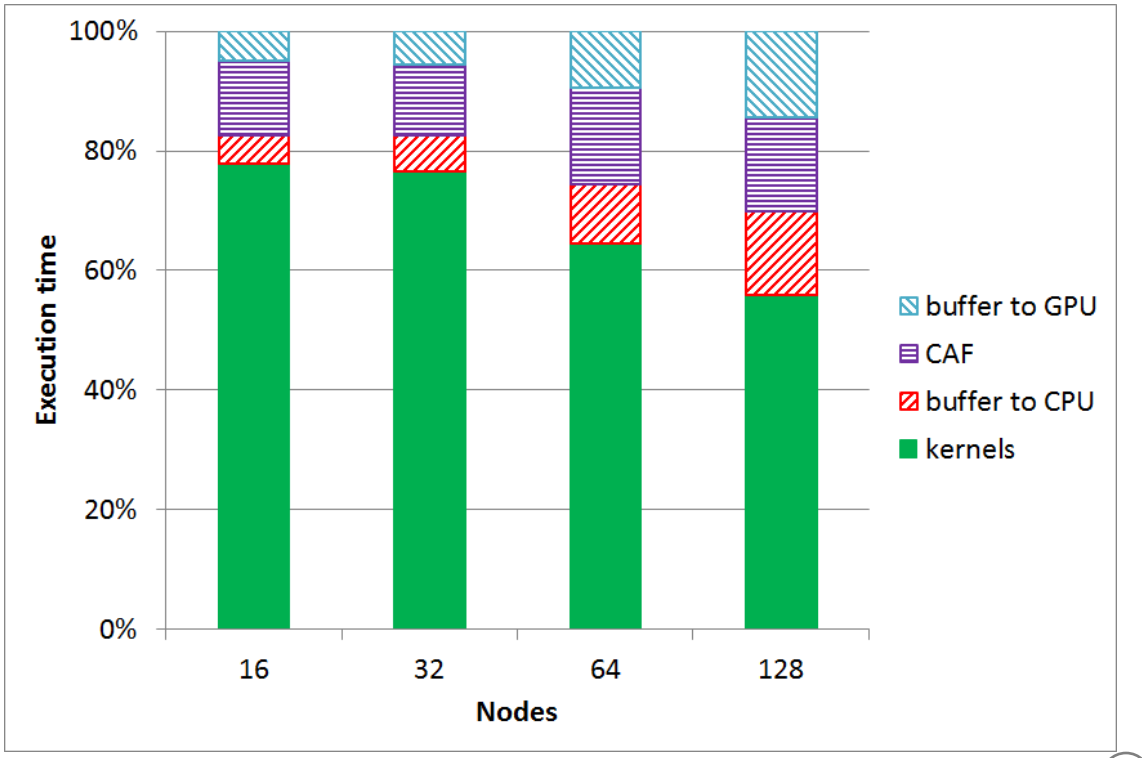
Himeno performance

- XK6 GPU is about 1.6x faster than XE6
- OpenACC async streams 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**



libsci_acc: LibSci for Accelerators how to get CPU&GPU cooperation

- **Provide basic libraries for accelerators, tuned for Cray**
- **Must be independent to OpenACC, but fully compatible**
- **Multiple use case support**
  - Get the base use of accelerators with no code change
  - Get extreme performance of GPU with or without code change
  - Extra tools for support of complex code
- **Incorporate the existing GPU libraries into libsci**
  - CUBLAS
  - Magma
  - Cray Implementation BLAS/LAPACK
- **Provide additional performance and usability**
  - OpenACC support
  - CUDA support
- **Maintain the Standard APIs where possible!**

Cray libsci_acc interfaces

- **Simple interface**

```
dgetrf(M, N, A, lda, ipiv, &info)
```

CPU

GPU + CPU

```
dgetrf(M, N, d_A, lda, ipiv, &info)
```

GPU

- **Device interface**

```
dgetrf_acc(M, N, d_A, lda, ipiv, &info)
```

GPU

- **CPU interface**

```
dgetrf_cpu(M, N, A, lda, ipiv, &info)
```

CPU

libsci_acc interaction with OpenACC

- If the rest of the code uses OpenACC, it's possible to use the library with directives.
- All data management performed by OpenACC.
- Calls the device version of dgemm.
- All data is in CPU memory before and after data region.

```
!$acc data copy(a,b,c)

!$acc parallel
!Do Something
!$acc end parallel

!$acc host_data use_device(a,b,c)

call dgemm_acc('n','n',m,n,k,&
                alpha,a,lda,&
                b,ldb,beta,c,ldc)

!$acc end host_data
!$acc end data
```

libsci_acc interaction with OpenACC

- libsci_acc is a bit smarter that this.
- Since ‘a,’ ‘b’, and ‘c’ are device arrays, the library knows it should run on the device.
- So just dgemm is sufficient.

```
!$acc data copy(a,b,c)

!$acc parallel
!Do Something
!$acc end parallel

!$acc host_data use_device(a,b,c)

call dgemm     ('n','n',m,n,k,&
                alpha,a,lda,&
                b,ldb,beta,c,ldc)

!$acc end host_data
!$acc end data
```

A large application performance breakdown

- **Comparing runs on 576 Cray XK6 nodes**
- **Different optimal configurations**
  - CPU: 48x48 = 2304 MPI, 4 OpenMP
  - CPU+GPU: 24x24 = 576 MPI, 16 OpenMP + CUDA
- **Performance comparison**
  - Kernel code on GPU is 3x faster than on CPU
  - MPI takes more time on the CPU version – 4x MPI ranks
  - MPI takes 30% of total time on CPU, 45% on the CPU+GPU version