CRAY
THE SUPERCOMPUTER COMPANY

Cray's Approach to Heterogeneous Computing

Roberto Ansaloni
roberto@cray.com

International Research Workshop
Advanced High Performance Computing Systems
Cetraro, June 27 - 29, 2011

Why heterogeneous computing in HPC ?

- Extrapolating current systems based on multi-core X86 CPUs will lead to unacceptably high power costs
- Multi-core CPUs are optimized for making single threads run fast, rather than many threads run power efficiently
- Heterogeneous nodes combining traditional multi-core CPUs with vector/SIMD accelerators hold promise to improve the power efficiency of HPC systems
- Another approach consists of supplying many more low-power cores on the node
- Both solutions present extreme programmability challenges !

HPC-ready accelerators today: GPU

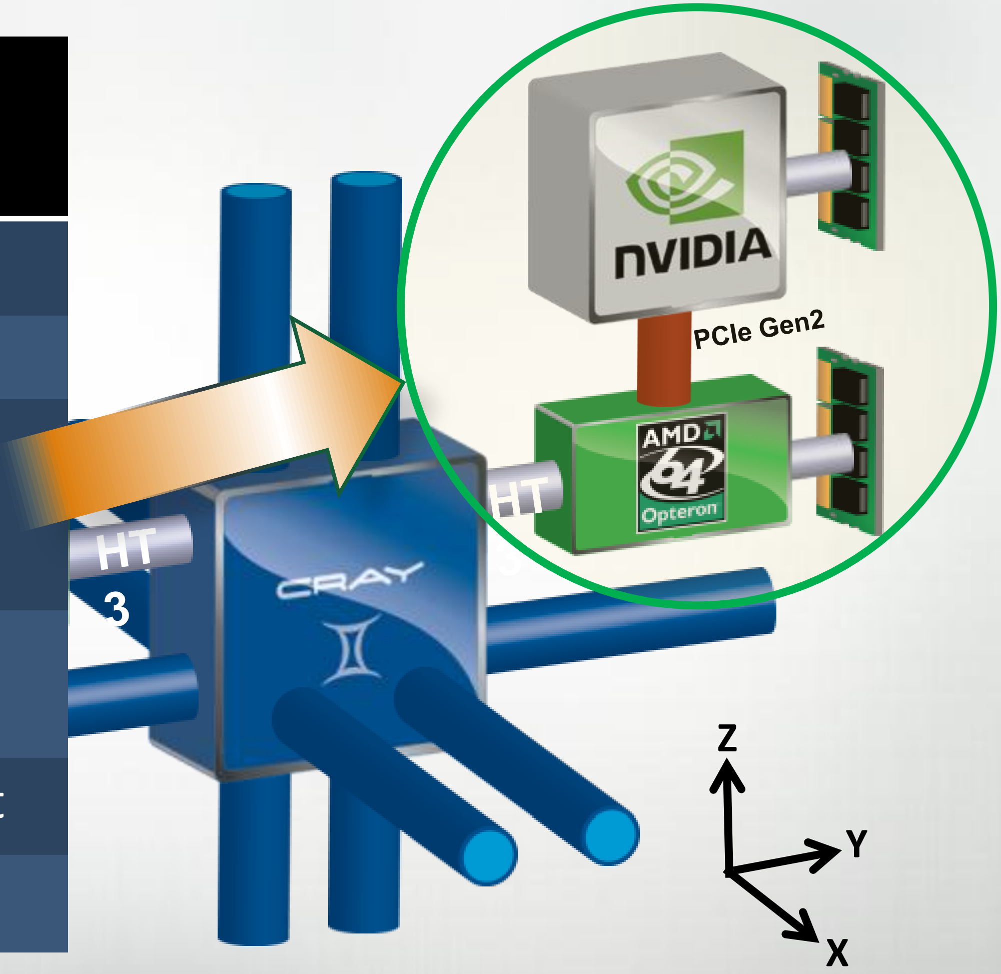
- NVIDIA Fermi^{TM} has made GPUs feasible for HPC
  - Robust error protection and strong DP FP, plus programming enhancements
- Expect GPUs to make continued and significant inroads into HPC
  - Compelling technical reasons + high volume market
- We are interested in large scale GPU configurations
  - Not really interested in single GPU results
  - Scaling GPU computing introduces new complexity

The Cray XK6

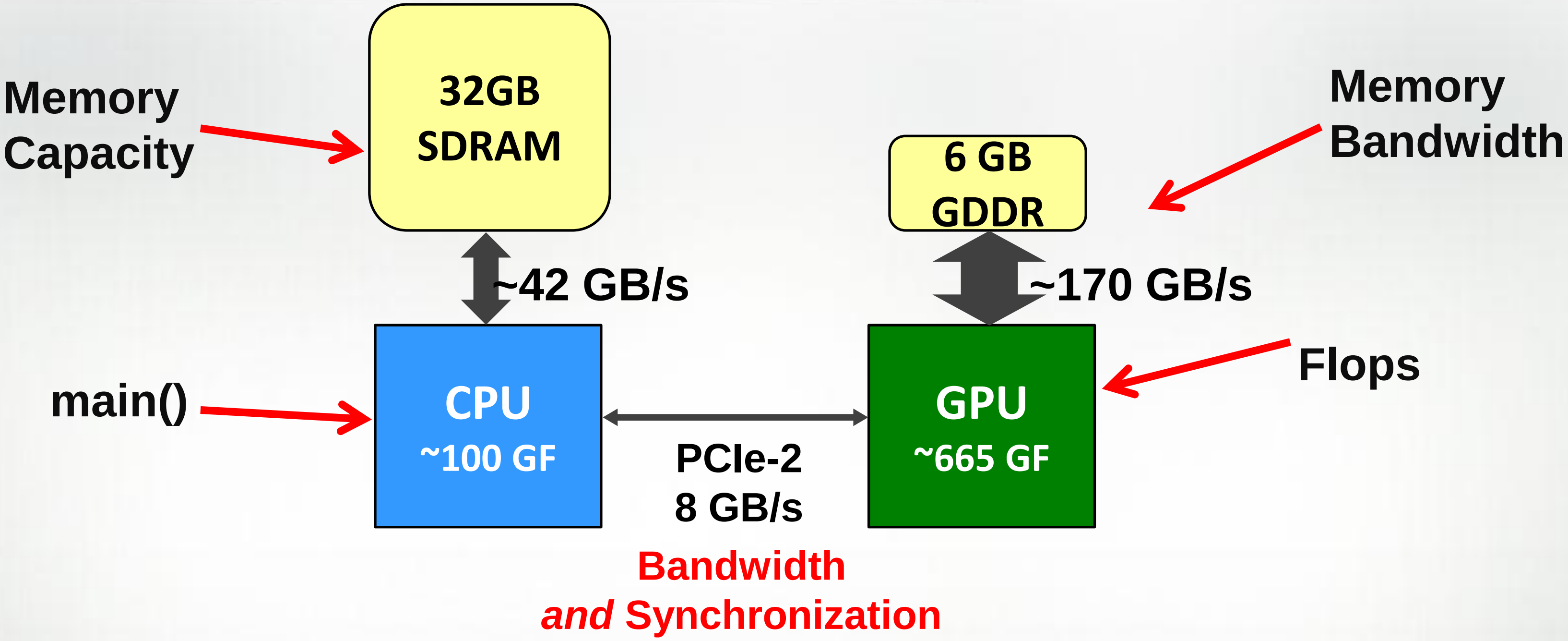
- Hybrid architecture: heterogeneous nodes combining CPU and GPU
    - CPU: AMD Opteron 6200 (Interlagos), 16 cores per socket
    - GPU: Nvidia Tesla X2090 (Fermi+)
- Gemini Interconnect
- Unified X86/GPU programming environment

Cray XK6 Compute Node

XK6 Compute Node Characteristics
AMD Series 6200 (Interlagos)
NVIDIA Tesla X2090
Host Memory 16 or 32GB 1600 MHz DDR3
NVIDIA Tesla X2090 Memory 6GB GDDR5 capacity
Gemini High Speed Interconnect
Upgradeable to future GPUs



Structural Issues with Accelerated Computing



- This is a short-lived situation
  - Solutions coming from several vendors (NVIDIA, AMD,…)
- Keep kernel data structures resident in GPU memory
  - Avoids copying b/w CPU and GPU; work on GPU-network communication
- May limit breadth of applicability over next 2-3 years

Programming Issues with Accelerated Computing

- Primary issues with programming for GPUs:
  - Learn new language/programming model
  - Maintain two code bases/lack of portability
  - Tuning for complex processor architecture (and split CPU/GPU structure)
- Need a single programming model that is portable across machine types and also forward scalable in time
  - Portable expression of heterogeneity and multi-level parallelism
  - Programming model and optimization should not be significantly difference for “accelerated” nodes and multi-core x86 processors
- Allow users to maintain a single code base

Unified X86/GPU programming environment

- Cray XK6 includes the first-generation Cray Unified X86/GPU Programming Environment
- Why is Cray putting so much effort into this?
  - Need to shield user from the complexity of dealing with heterogeneity
  - High level language with good compiler and runtime support
  - Optimized libraries for heterogeneous multicore processors
- It will support three classes of users:
  1. "hardcore" GPU programmers with existing CUDA ports
  2. users with parallel codes and OpenMP experience, but less GPU knowledge
  3. users with serial codes looking for portable parallel performance with and without GPUs

OpenMP for Accelerators

- An open standard is the most attractive for developers
  - portability; multiple compilers for debugging; permanence
- An established standards committee is better than a new body
  - Subcommittee of OpenMP ARB, aiming for OpenMP 4.0
  - includes most major vendors (PGI, CAPS, Intel, IBM... + other interested parties (e.g. EPCC)
- Co-chaired by Cray (James Beyer)
- Cray is an enthusiastic supporter
  - CCE is first full implementation
  - Fortran, C, C++

A simple loop on the GPU

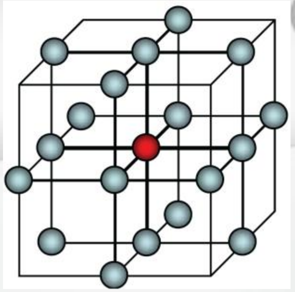
```
!$omp acc_region_loop
DO j = 1,M
  DO i = 2,N
    c(i,j) = a(i,j) + b(i,j)
  ENDDO
ENDDO
!$omp end acc_region_loop
```

- Compiler does the work:
- Data movement
  - allocates/frees GPU memory
  - moves of data to/from GPU
- Loop schedule: spreading loop iterations over PEs of GPU
  - division of iterations between SIMT/MIMD units of GPU
- Cache usage
  - Explicit use of GPU shared memory for reused data
    - automatic caching (e.g. NVIDIA Fermi) important
- Tune default behaviour with optional clauses on directives

The Himeno Benchmark

- 3D Poisson equation
  - 19-point stencil
  - Highly memory intensive, memory bandwidth bound
- Fortran, C, MPI and OpenMP implementations available from http://accc.riken.jp/HPC_e/himenobmt_e.html
- Strong scaling benchmark
  - Tests on XL configuration: 1024 x 512 x 512
- NVIDIA paper on GPU CUDA implementation
  - Phillips, E.H.; Fatica, M.;
    Implementing the Himeno benchmark with CUDA on GPU clusters
    IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 2010

The Jacobi computational kernel



- 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
```

Distributed implementation

- The outer loop is performed a fixed number of times
- Jacobi kernel is executed and new pressure array work2 and control value wgosa are computed
- The array is updated with the new pressure values
- The halo region values are exchanged between neighbor PEs
- Send and receive buffers are used
- The global sum of the control values is computed with an Allreduce operation across all the PEs

```
DO loop = 1, nn
   compute Jacobi kernel and get wrk2
   and wgosa

   copy back wrk2 into p

   pack halo region data from p into
   send buffers

   perform halo exchange with neighbor
   PEs

   unpack halo region data from recv
   buffers into p

   Allreduce to get global sum of wgosa
   across all the PEs
ENDDO
```

Himeno on the Cray XK6

- Several versions tested, with communication implemented in MPI or Fortran coarrays
- GPU version using early pre-release implementation of OpenMP Accelerator directives
- Arrays reside permanently in the GPU memory
- Data transfers between host and GPU are due to:
  - Communication buffers for the halo exchange
  - Control value
- Compare Cray XK6 timings with best Cray XE6 results (hybrid MPI/OpenMP)
  - Same number of nodes fully utilized: 1 GPU vs 24 CPU cores

Allocating arrays on the GPU

- Arrays are allocated on the GPU memory in the main program with the *acc_data* directive
- In the subroutines the *acc_data* directive is replicated with the *present* clause, to use the data already present in the GPU memory and avoid extra allocations

```
PROGRAM himenobmtxp
...
!$omp acc_data acc_shared (          &
!$omp&  p,a,b,c,wrk1,wrk2,bnd,       &
!$omp&  sendbuffx_up,sendbuffx_dn,   &
!$omp&  sendbuffy_up,sendbuffy_dn,   &
!$omp&  sendbuffz_up,sendbuffz_dn)
...
!$omp end acc_data

SUBROUTINE jacobi(nn,gosa)
!$omp acc_data present (             &
!$omp&  p,a,b,c,wrk1,wrk2,bnd,       &
!$omp&  sendbuffx_up,sendbuffx_dn,   &
!$omp&  sendbuffy_up,sendbuffy_dn,   &
!$omp&  sendbuffz_up,sendbuffz_dn)
```

Preliminary: directive syntax may change – functionality will be the same

Jacobi kernel on the GPU

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

```
DO loop=1,nn
  gosa=0.0
  wgosa=0.0
!$omp acc_region_loop              &
!$omp&  private(s0,ss)             &
!$omp&  reduction(+:wgosa)         &
!$omp&  num_pes(2:256)
  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
```

Preliminary: directive syntax may change – functionality will be the same

Halo region buffers

- Halo values are extracted from the wrk2 array and packed into the send buffers, on the GPU
- A global *acc_region* is specified and buffers in the X, Y, and Z directions are packed within *acc_loop* blocks
- The send buffers are copied to host memory with *acc_update*
- In the same way, after the halo exchange, the recv buffers are transferred to the GPU memory and used to update the array p

```
!$omp acc_region
!$omp acc_loop
DO j = 2,jmax-1
  DO i = 2,imax-1
    sendbuffz_dn(i,j)= wrk2(i,j,2)
    sendbuffz_up(i,j)= wrk2(i,j,kmax-1)
  ENDDO
ENDDO
!$omp end acc_loop
 ...
!$omp acc_loop
!$omp end acc_loop
!$omp end acc_region

!$omp acc_update &
!$omp&  host(sendbuffz_dn,sendbuffz_up)
```

Preliminary: directive syntax may change – functionality will be the same

Fortran 2008 Coarray implementation

- Coarrays are used to perform the halo exchange
- A non-blocking communication is triggered by the *pgas defer_sync* directive
- In this way the programmer is responsible of the data synchronization
- By setting the sync point as far as possible, communication can be overlapped to CPU or GPU activity
- In this case the array p update from wrk2, on the GPU, can be overlapped to the halo exchange

```
recvbuffz_up(:,:)[myx,myy,myz-1] =
sendbuffz_dn(:,:)
 ...
!$omp acc_region_loop
DO k = 2,kmax-1
  DO j = 2,jmax-1
    DO i = 2,imax-1
      p(i,j,k) = wrk2(i,j,k)
    ENDDO
  ENDDO
ENDDO
!$omp end acc_region_loop
sync memory
!$omp acc_update &
!$omp& acc(recvbuffz_dn,recvbuffz_up)
```

Preliminary: directive syntax may change – functionality will be the same

Ease of use: total number of lines

Original Himeno MPI-Fortran code:	629
C /CUDA code:	823
Version with coarrays and accelerator directives:	554
Total number of accelerator directives:	24

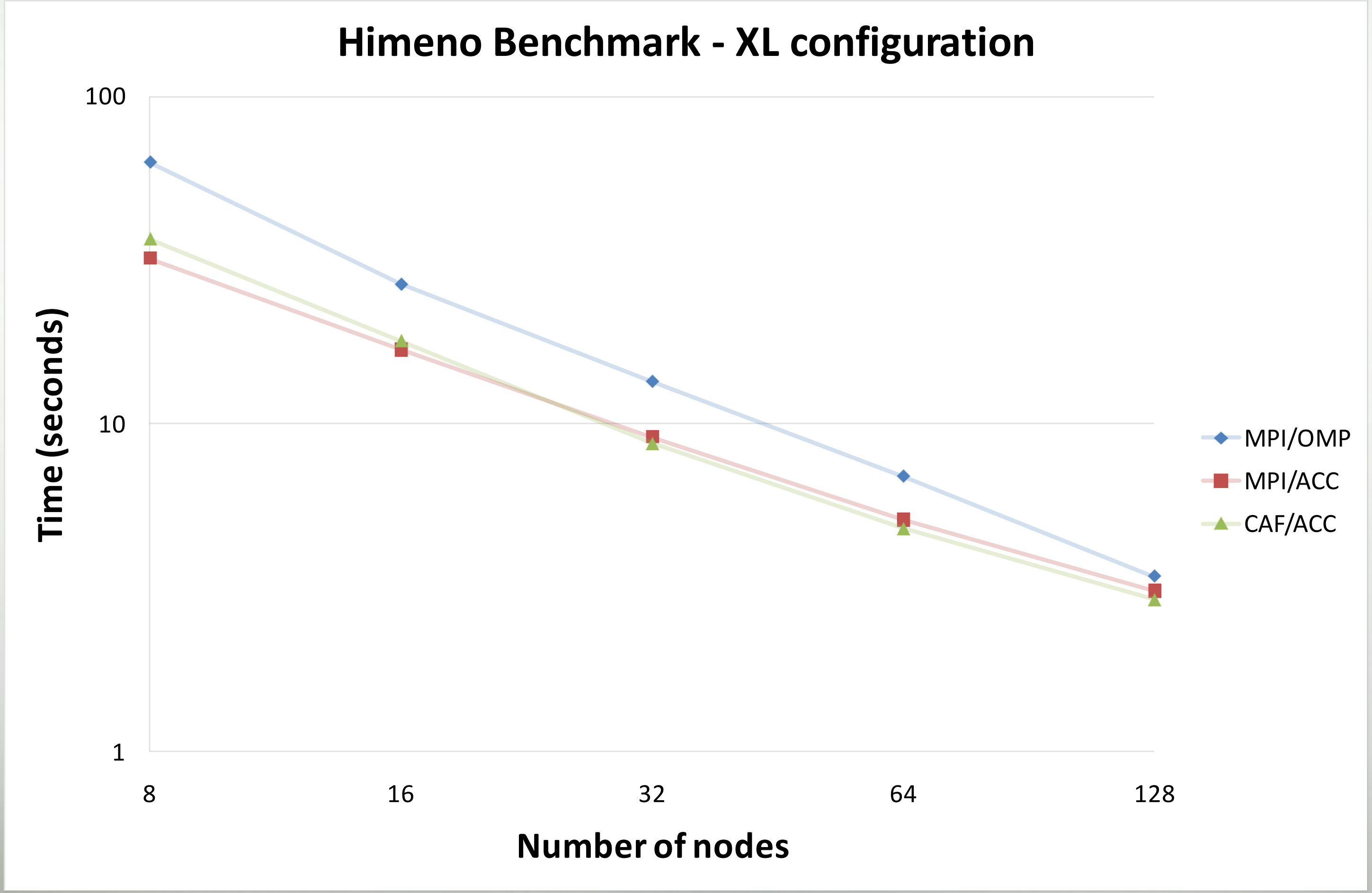
Benchmarking the code

- XK6 configuration:
  - MC-12 2.1GHz CPU cores, 12 cores per node
  - Tesla X2090 GPU
  - Running with 1 PE (GPU) per node
  - Himeno case XL needs at least 8 XK6 nodes

- XE6 configuration:
  - MC-12 2.1 GHz nodes, 24 cores per node
  - Running on fully packed nodes: all cores used
  - Depending on the number of nodes, 1-6 OpenMP threads per PE are used

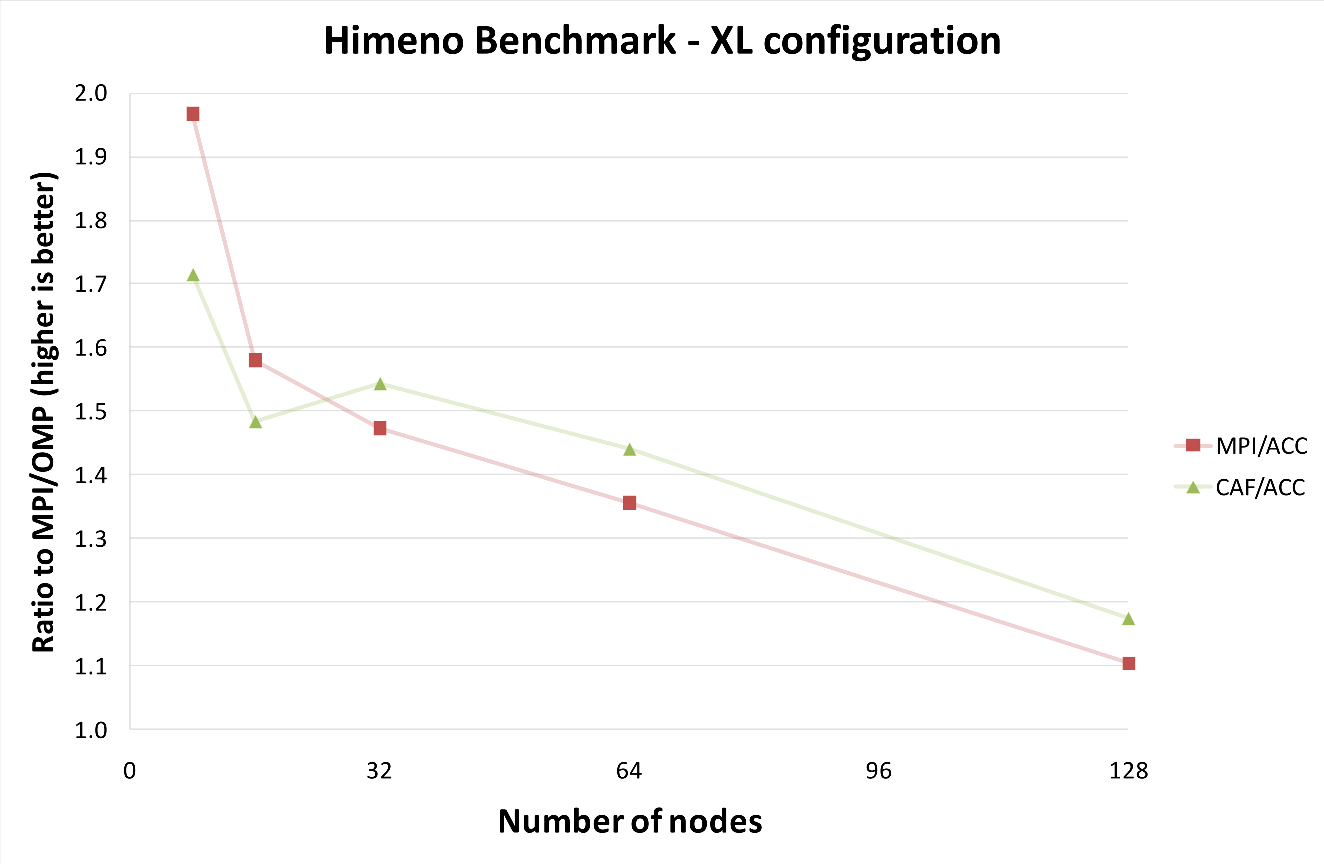
Himeno Timings

- Tha accelerated code on the XK6 outperforms the XE6
- Larger gap on small number of nodes
- CAF communication is more efficient than MPI



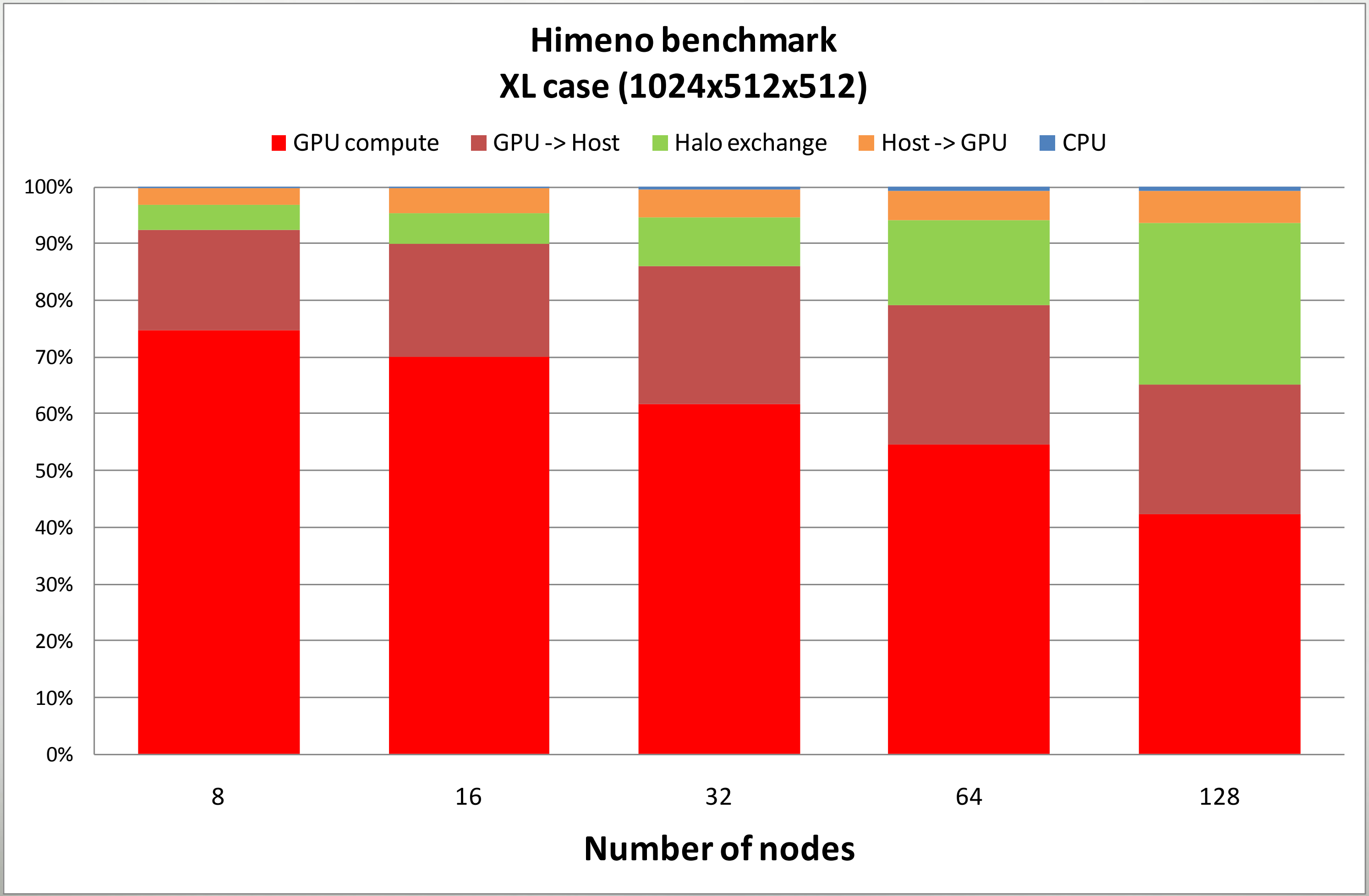
XK6 (heterogeneous) ratio to XE6 (CPU)

- XK6 is always faster
  - On 16 nodes the CPU code gets a superlinear boost due to cache effect
- On 8 nodes the GPU is about 200% faster than the CPU
- On 128 nodes the GPU is about 20% faster than the CPU



MPI/ACC code breakdown (craypat!)

- The host/GPU transfers take a significant amount of time
  - this code would benefit from an efficient direct GPU-GPU communication
- On 128 nodes less than 40% of the time is spent in the GPU compute kernel



Very Early XK6 Performance Results

Benchmark	Programming Model	Unit of Computation	Performance vs. *Dual* Opteron Node
ICCG	OpenMP	Single Node	1.8
LBM	OpenMP/MPI	Multiple Nodes	2.1
CFD	OpenMP	Single Node	5
S3D kernels	OpenMP/MPI	Multiple Nodes	1.5 - 6
CG	OpenMP	Single Node	1.1
Himeno	OpenMP/MPI	Multiple Nodes	1.2 - 2.0
SWIM	OpenMP	Node	1.7

- GPU performance (currently not using Opteron)
- Alpha version tools and results (very much a work in progress)
- MOST EXCITING RESULT: Tuning for GPU has significantly improved the all-CPU version in many cases, while retaining a compatible programming model

Long Term Prognosis (Steve Scott)

- Before the end of this decade, we won’t think of this as “accelerated computing” any more.
  - It will just be how computing is done.
- The structural issues with today’s GPU systems will be gone.
  - Hybrid multicore with shared memory hierarchy.
- Programming environments for hybrid multicore will be much improved
  - The user’s job will be to expose parallelism and convey information about locality
  - The compiler and runtime will map onto the hardware
  - One version of code
- The next few years will be fun and challenging
  - Will take significant work to tune applications for extreme scale
  - Doesn’t have to be specific to accelerators; will pay off generally

Thank You. Questions?

CRAY
THE SUPERCOMPUTER COMPANY

Acknowledgments:
Alistair Hart, John Levesque, Bill Long, Steve Scott, Luiz DeRose, James Beyer