A High Level Programming Environment for Accelerated Computing

Luiz DeRose
Sr. Principal Engineer
Programming Environments Director
Cray Inc.

Outline

- Motivation
- Cray XK6 Overview
- Why a new programming model
- OpenACC overview
- The Cray programming environment for accelerated computing
- Case study
- Conclusions

The Exascale furrow is a hard one to plough...

Seymour Cray famously once asked if you would rather plough a field with two strong oxen or five-hundred-and-twelve chickens.

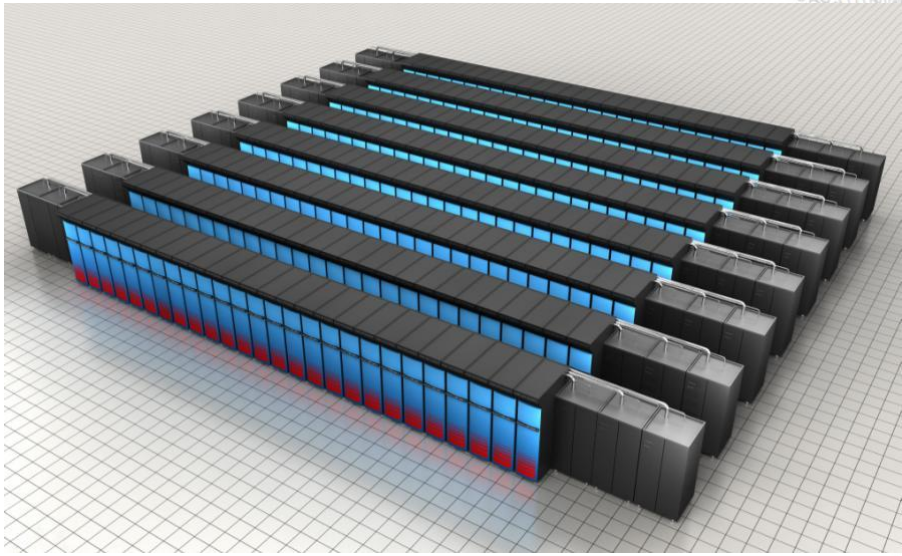
"Since then, the question has answered itself: power restrictions have driven CPU manufacturers away from "oxen" (powerful single-core devices) towards multi- and many-core "chickens".

"An exascale supercomputer will take this a step further, connecting tens of thousands of many-core nodes.

"Application programmers face the challenge of harnessing the power of tens of millions of threads."

EPCC News, issue 70 (Autumn 2011)

Upcoming Heterogeneous Multi Petaflop Systems



Blue Waters: Sustained Petascale Performance

- Production Science at Full Scale
- 235 XE Cabinets + 30 XK Cabinets
  - > 25K compute nodes
- 11.5 Petaflops
- 1.5 Petabytes of total memory
- 25 Petabytes Storage
  - 1 TB/sec IO
- Cray's scalable Linux Environment
- HPC-focused GPU/CPU Programming Environment

Titan: A "Jaguar-Size" System with GPUs

- 200 cabinets
- 18,688 compute nodes
- 25x32x24 3D torus (22.5 TB/s global BW)
- 128 I/O blades (512 PCIe-2 @ 16 GB/s bidir)
- 1,278 TB of memory
- 4,352 sq. ft.
- 10 MW

Three Levels of Parallelism Required

- Developers will continue to use MPI between nodes or sockets
- Developers must address using a shared memory programming paradigm on the node
- Developers must vectorize low level looping structures
- While there is a potential acceptance of new languages for addressing all levels directly. Most developers cannot afford this approach until they are assured that the new language will be accepted and the generated code is within a reasonable performance range

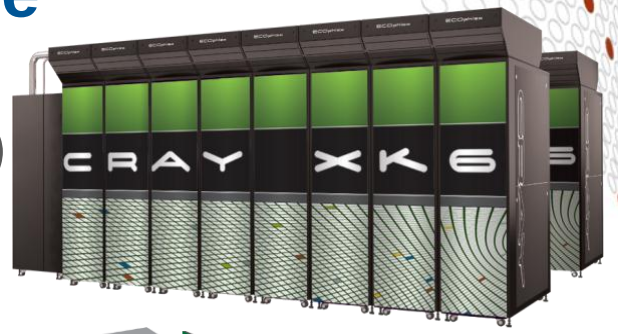
Cray Vision for Accelerated Computing

- **Most important hurdle** for widespread adoption of accelerated computing in HPC **is programming difficulty**
  - Need a single programming model that is **portable across machine types**
    - **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**

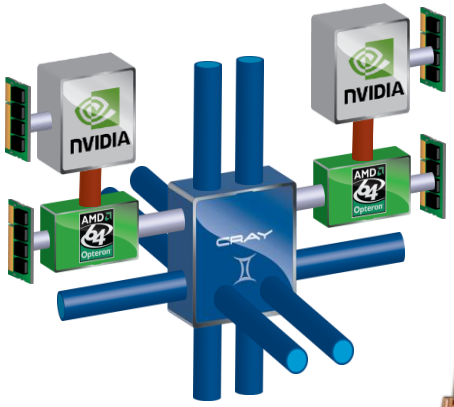
- **Cray's approach to Accelerator Programming is to provide an ease of use tightly coupled high level programming environment with compilers, libraries, and tools that can hide the complexity of the system**

- **Ease of use is possible with**
  - Compiler making it **feasible for users** to write applications in **Fortran, C,** and **C++**
  - Tools to help users port and optimize for hybrid systems
  - Auto-tuned scientific libraries

"Accelerating the Way to Better Science"



- **Cray has announced the Cray XK6 (May'11)**
  - NVIDIA Fermi X2090 GPU
    - Upgradable to Kepler
  - AMD Interlagos CPU
  - Cray Gemini interconnect
    - high bandwidth/low latency scalability
  - Fully compatible with Cray XE6 product line
  - Fully upgradeable from Cray XT/XE systems



Unified X86/GPU Programming Environment

- **The Cray XK6 includes the first-generation of the Cray Unified X86/GPU Programming Environment**

- **Why is Cray putting so much effort into this?**
  - Opens up GPU computing to a larger user base
  - A **good programming environment narrows the gap** between observed and achievable performance

- **It supports three classes of users:**
  1. “Hardcore" GPU programmers with existing CUDA ports
  2. Users with parallel codes, ideally with some OpenMP experience, but less GPU knowledge
  3. Users with serial codes looking for portable parallel performance with and without GPUs

Programming for a Node with Accelerator

- **Fortran, C, and C++ compilers**
  - **OpenACC directives to drive compiler optimization**
  - Compiler does the “heavy lifting” to split off the work destined for the accelerator and perform the necessary data transfers
  - Compiler optimizations to take advantage of accelerator and multi-core X86 hardware appropriately
  - Advanced users can mix CUDA functions with compiler-generated accelerator code
  - **Parallel Debugger support** with DDT or TotalView
- **Cray Reveal, built upon an internal compiler representation of the application (the Cray Compiler Program Library)**
  - Source code browsing tool that provides interface between the user, the compiler, and the performance tool
    - **Scoping tool** to help users port and optimize applications
    - **Performance measurement and analysis** information for porting and optimization
- **Scientific Libraries support**
  - Auto-tuned libraries (using Cray Auto-Tuning Framework)

OpenACC Accelerator Programming Model

- **Why a new model? There are already many ways to program:**
  - CUDA and OpenCL
    - All are quite low-level and closely coupled to the GPU
  - PGI CUDA Fortran
    - Still CUDA just in a better base language
  - PGI accelerator directives, CAPS HMPP
    - First steps in the right direction – Needed standardization
- **User needs to write specialized kernels:**
  - **Hard** to write and debug
  - **Hard** to optimize for specific GPU
  - **Hard** to update (porting/functionality)
- **OpenACC Directives provide high-level approach**
  - **Simple programming model for heterogeneous systems**
  - **Easier to maintain/port/extend code**
    - The same source code can be compiled for multicore CPU
  - Based on the work in the OpenMP Accelerator Subcommittee
    - Proposed to the OpenMP Language Committee
      - Subcommittee of OpenMP ARB, aiming for OpenMP 4.0
  - Possible performance sacrifice
    - A small performance gap is acceptable (do you still hand-code in assembler?)
    - Goal is to provide at least 90% of the performance obtained with hand coded CUDA
      - Already seeing this in many cases, more tuning ongoing

OpenACC Accelerator Directives

- **An open standard is the most attractive for developers**
  - **Portability**; multiple compilers for debugging; permanence
  - Helps programmer tools proliferation
  - Provides faster time to adoption of accelerators

- **What is OpenACC based on?**
  - It is **based on the work of Cray and PGI in their compilers that support accelerators today**
  - It is also based on the extensions proposed to the OpenMP Accelerator subcommittee

- **What is the difference between OpenACC and PGI/Cray accelerator directives?**
  - Primarily syntax and a **common subset of features**
    - Individual compiler products may have other capabilities beyond the standard

- **Will OpenACC run on NVIDIA GPUs with CUDA?**
  - Yes – but OpenACC does not require NVIDIA GPUs
    - Some programmers may wish to develop easy code using directives, and take advantage of already developed CUDA code

- **Will OpenACC run on AMD GPU's and on top of OpenCL?**
  - It could, it requires implementation, there is no reason why it couldn't

How About OpenMP Extensions for Accelerators

- **Subcommittee of OpenMP ARB, aiming for OpenMP 4.0**
  - Co-chaired by Cray
  - Includes most major vendors
    - Cray, NVIDIA, PGI, CAPS, Intel, IBM ... + other interested parties

- **OpenACC is based on the common work in the OpenMP accelerator subcommittee**

- **We hope to encourage OpenMP to leverage this work in the development of that standard**
  - This is **an opportunity to get real developer feedback** on OpenACC that should lead to a more complete and robust standard for OpenMP

Accelerator directives

- **Modify original source code with directives**
  - Non-executable statements (comments, pragmas)
    - Can be ignored by non-accelerating compiler
    - CCE -hnoacc also supresses compilation
  - Sentinel: !$acc
  - Fortran:
    - Usually paired with !$acc end *
  - C/C++:
    - Structured block {...} avoids need for end directives
  - Continuation to extra lines allowed

```
! Fortran example
!$acc *
<structured block>
!$acc end *
```

```
/* C/C++ example */
#pragma acc *
{structured block}
```

A First Example: Execute a region of code on the GPU

- **Compiler does the work:**
  - **Identifies parallel loops within the region**
  - Determines the kernels needed
  - **Splits the code into accelerator and host portions**
  - Workshares loops running on accelerator
    - Make use of MIMD and SIMD style parallelism
  - **Data movement**
    - allocates/frees GPU memory at start/end of region
    - **moves data to/from GPU**

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

write-only

read-only

- User can tune default behavior with optional directives and clauses
- Loop schedule: spreading loop iterations over PEs of GPU

Parallelism	NVIDIA GPU	SMT node (CPU)
gang:	a threadblock	CPU
worker:	warp (32 threads)	CPU core
vector:	SIMT group of threads	SIMD instructions (SSE, AVX)

A First OpenACC Program: "Hello World"

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

- Two accelerator parallel regions
  - Compiler creates two kernels
    - Loop iterations automatically divided across gangs, workers, vectors
    - Breaking parallel region acts as barrier
  - First kernel initializes array
    - Compiler will determine copyout(a)
  - Second kernel updates array
    - Compiler will determine copy(a)
  - Breaking parallel region=barrier
    - No barrier directive (global or within SM)

- Array a(:) unnecessarily moved from and to GPU between kernels
  - "data sloshing"
- Code still compile-able for CPU

A second version

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

- Now added a **data** region
  - Specified arrays only moved at boundaries of data region
  - Unspecified arrays moved by each kernel
  - No compiler-determined movements for data regions
- Data region can contain host code and accelerator regions
- Copies of arrays independent

- No automatic synchronization of copies within data region
  - User-directed synchronization via **update** directive
- Code still compile-able for CPU

Directive Clauses

- **Data clauses:**
  - copy, copyin, copyout, create
    - e.g. copy moves data "in" to GPU at start of region and "out" to CPU at end
    - supply list of arrays or array sections
      - Fortran use standard array syntax (":" notation)
      - C/C++ use extended array syntax [start:length]
  - present: share GPU-resident data between kernels
  - present_or_copy [in,out]
    - use data if already resident, otherwise move the data
- **Tuning clauses:**
  - num_gangs, vector_length, collapse...
    - optimize GPU occupancy, register and shared memory usage, loop scheduling...
- **Some other important clauses:**
  - async: Launch accelerator region asynchronously
    - Allows overlap of GPU computation/PCI transfers with CPU computation/network

Sharing GPU data between subprograms

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

```
SUBROUTINE double_array(b)
  INTEGER :: b(N)
!$acc parallel loop present_or_copy (b)
  DO i = 1,N
   b(i) = double_scalar(b(i))
  ENDDO
!$acc end parallel loop
END SUBROUTINE double_array
```

```
INTEGER FUNCTION double_scalar(c)
  INTEGER :: c
  double_scalar = 2*c
END FUNCTION double_scalar
```

- One of the kernels now in subroutine (maybe in separate file)
  - CCE supports function calls inside **parallel** regions
    - Compiler will automatically inline
- The **present** clause uses version of b on GPU without data copy
  - Can also call double_array() from outside a data region
    - Replace **present** with **present_or_copy** (can be shortened to **pcopy**)
- Original calltree structure of program can be preserved

CUDA Interoperability

```
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
  - CUDA kernel written as usual
  - Or use same mechanism to call existing CUDA library

Reveal

New code restructuring and analysis assistant…

Uses both the performance toolset and CCE's program library functionality to provide static and runtime analysis information

Assists user with the code optimization phase by correlating source code with analysis to help identify which areas are key candidates for optimization

Key Features

Annotated source code with compiler optimization information

- Provides feedback on critical dependencies that prevent optimizations

Scoping analysis

- Identifies shared, private and ambiguous arrays
  - Allows user to privatize ambiguous arrays
  - Allows user to override dependency analysis

Source code navigation

- Uses performance data collected through CrayPat

Cray Performance Tools

- **Scaling (running big jobs with a large number of GPUs)**
  - **Results** summarized and **consolidated** in one place
- **Statistics for the whole application**
  - Performance **statistics mapped back** to the user **source** by line number
  - Performance statistics grouped by accelerator directive
  - Single report can include **statistics for both the host and the accelerator**
- **Single tool for GPU and CPU performance analysis**
  - Performance statistics
    - Includes accelerator time, host time, and amount of data copied to/from the accelerator
  - Kernel level statistics
  - Accelerator hardware counters

Types of Statistics

- **Loop work estimates**
  - Provide information to identify important loops
- **Performance statistics**
  - Includes accelerator time, host time, and amount of data copied to/from the accelerator
- **Accelerator hardware counters**
  - Hardware counters on the accelerator itself.
    - On NVIDIA Fermi GPUs, there are about 50 available counters
- **Kernel level statistics**
  - Includes stats about grid size, block size, and occupancy

Cray Scientific Libraries Approach

1. **Maintain thorough performance evaluation of the state-of-the-art**

2. **Where we can make a difference provide something**
   - Difference in terms of functionality (e.g. libsci advanced)
   - Performance difference

3. **Where possible drive development from applications**

What is Cray Libsci_acc?

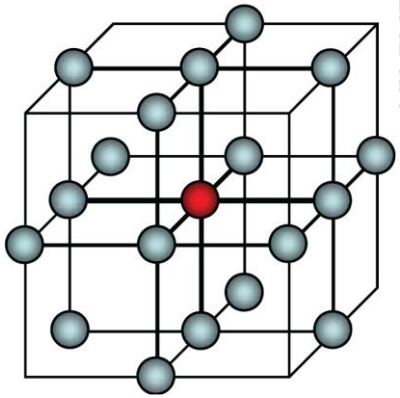
- **Selects best GPU kernel for the current task based on:**
  - Problem, Problem Size, Data Size
- **Selects best Kernel from:**
  - Cray tuned kernels (ATF)
  - cuBlas, magmaBlas
  - Other available sources
- **Provides two sets of interfaces to be used in difference scenarios with *minimized code modifications***
  - Basic Interface:
    - Data copy is automatic
    - GPU or CPU execution placement is automatic
    - Automatic Memcpy optimizations
      - Copy only necessary data (submatrix copy, basic interface)
  - Advanced Interface:
    - Data placement done by user
    - CCE Integration

Third Party Integration

- **In addition to the Cray Differentiated Programming Environment for GPUs, the following third party components are also available for the Cray XK6:**

- Compilers
  - NVIDIA C and C++
  - PGI Fortran, C, and C++
  - CAPS
- Libraries
  - CUDA Runtime support libraries
  - NVIDIA Thread Storage libraries
  - NVIDIA GPU-accelerated BLAS
  - NVIDIA GPU-accelerated FFT
  - MAGMA
- Tools
  - Environment setup
    - Modules
  - Debuggers
    - NVIDIA debugger
    - TotalView
    - DDT
  - Performance Tools
    - CUDA Visual Profiler
    - OpenCL Visual Profiler

Example: 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

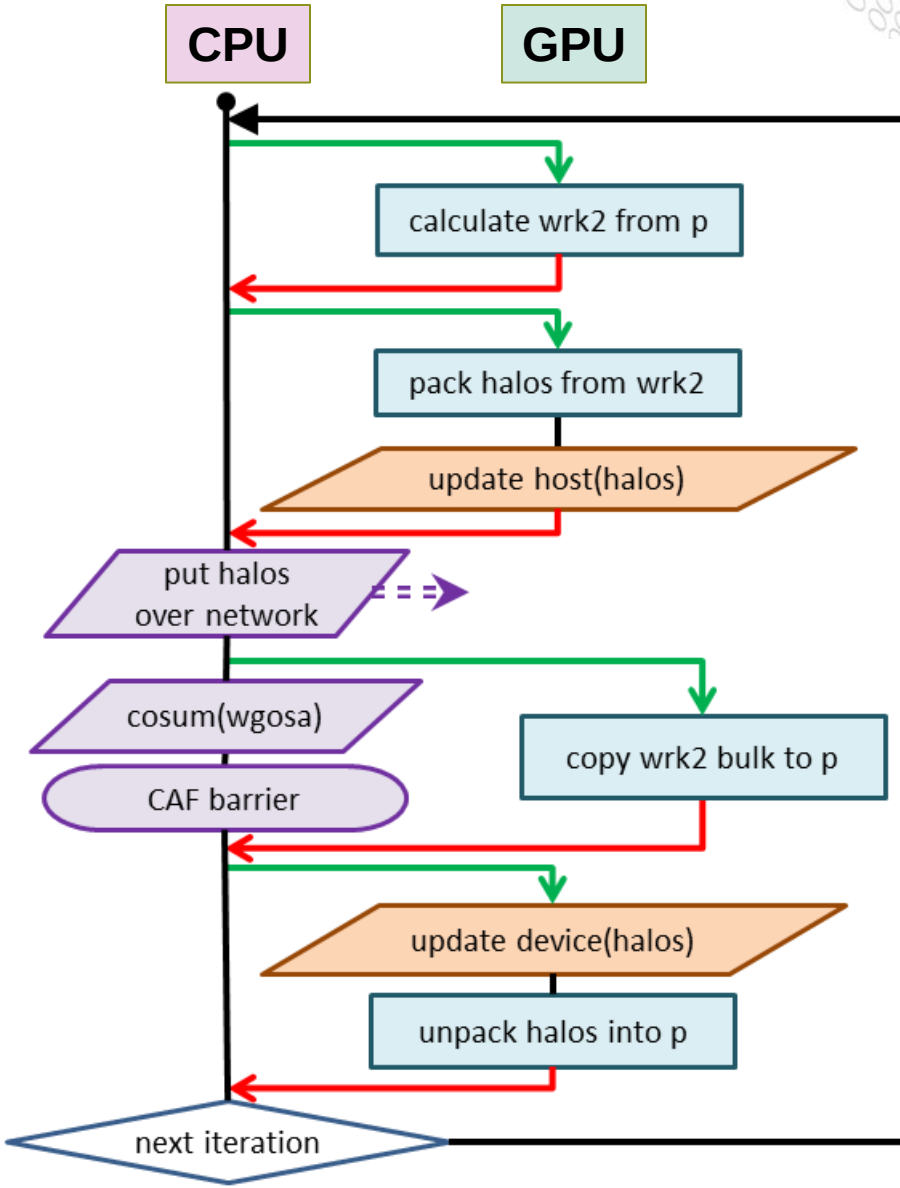
```
DO loop = 1, nn

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

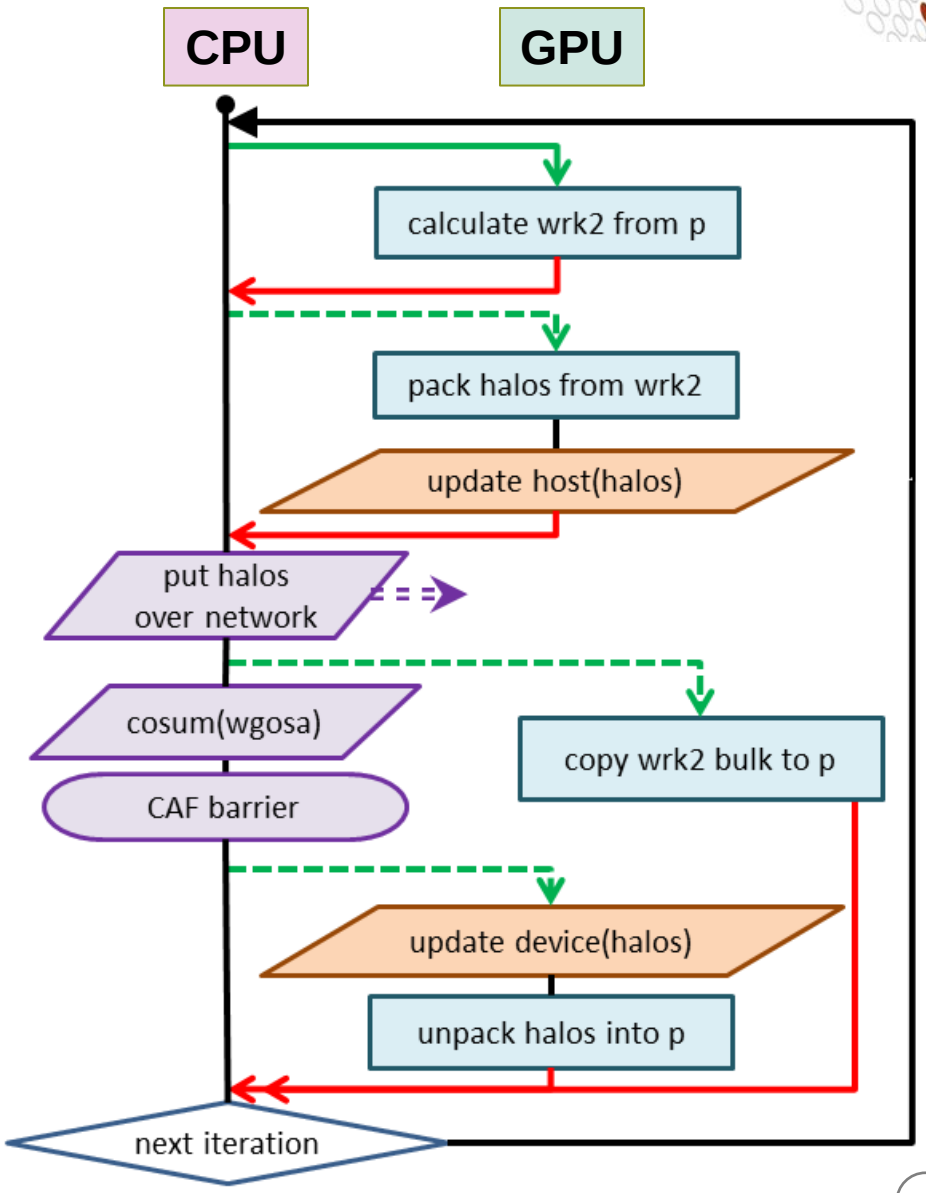
The Himeno GPU code structure

- **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



Using asynchronous streams

- **Async buffer handling**
  - Packing/unpacking multiple buffers
  - Overlapping packing and host/device transfers
- **Further testing possible**
  - Overlapping/pipelining CAF remote put with host/device transfers ?
  - Pinned memory allocation for the halo buffers ?



Allocating arrays on the GPU

- Arrays are allocated on the GPU memory in the main program with the **data** directive
- In the subroutines the **data** directive is replicated with the **present** clause, to use the data already present in the GPU memory and avoid extra allocations
- Since **present** clause is used, no **copy*** clauses are used, and data transfers to/from host are implemented by **update** directives

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


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

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
```

Halo region buffers

- Halo values are extracted from the wrk2 array and packed into the send buffers, on the GPU
- A global `parallel` region is specified and buffers in the X, Y, and Z directions are packed within `loop` blocks
- The send buffers are copied to host memory with `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

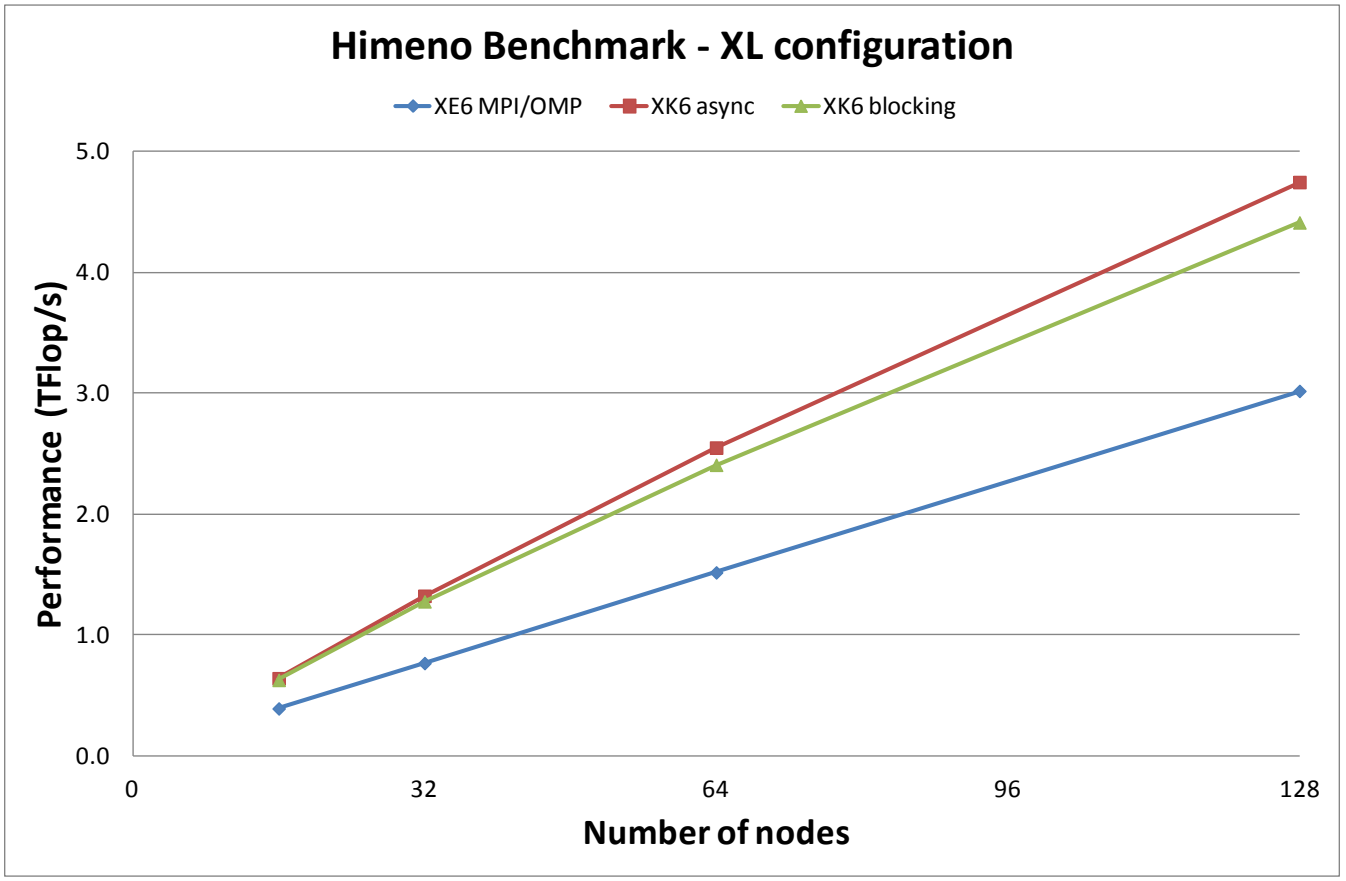
```
!$acc parallel
!$acc loop
DO j = 2,jmax-1
  DO i = 2,imax-1
    sendbuffz_dn(i,j)= wrk2(i,j,2)
    sendbuffz_up(i,j)= wrk2(i,j,kmax-1)
  ENDDO
ENDDO
!$acc end loop
 ...
!$acc loop
 ...
!$acc end loop
!$acc end parallel

!$acc update host
!$acc&       (sendbuffz_dn,sendbuffz_up)
```

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



Summary

- **Hybrid multicore has arrived and is here to stay**
  - Fat nodes are getting fatter
  - GPUs have leapt into the top500 and accelerated nodes
- **Programming accelerators efficiently is hard**
  - Need a high level programming environment
    - Cray Compilation Environment (CCE) focused on ease-of-use
      - OpenACC support
      - "Program Library" provides application specific repository for information for compiler and tools
    - Cray Reveal
      - Assists user in understanding their code and taking full advantage of SW and HW system
    - Cray Performance Analysis Toolkit
      - Single tool for GPU and CPU performance analysis with statistics for the whole application
    - Cray Auto-Tuning Libraries
      - Getting performance from the system … no assembly required

A High Level Programming Environment for Accelerated Computing

Questions / Comments

Thank You