OpenACC: A High Level Programming Model for Accelerated Computing

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

Supercomputing Leadership

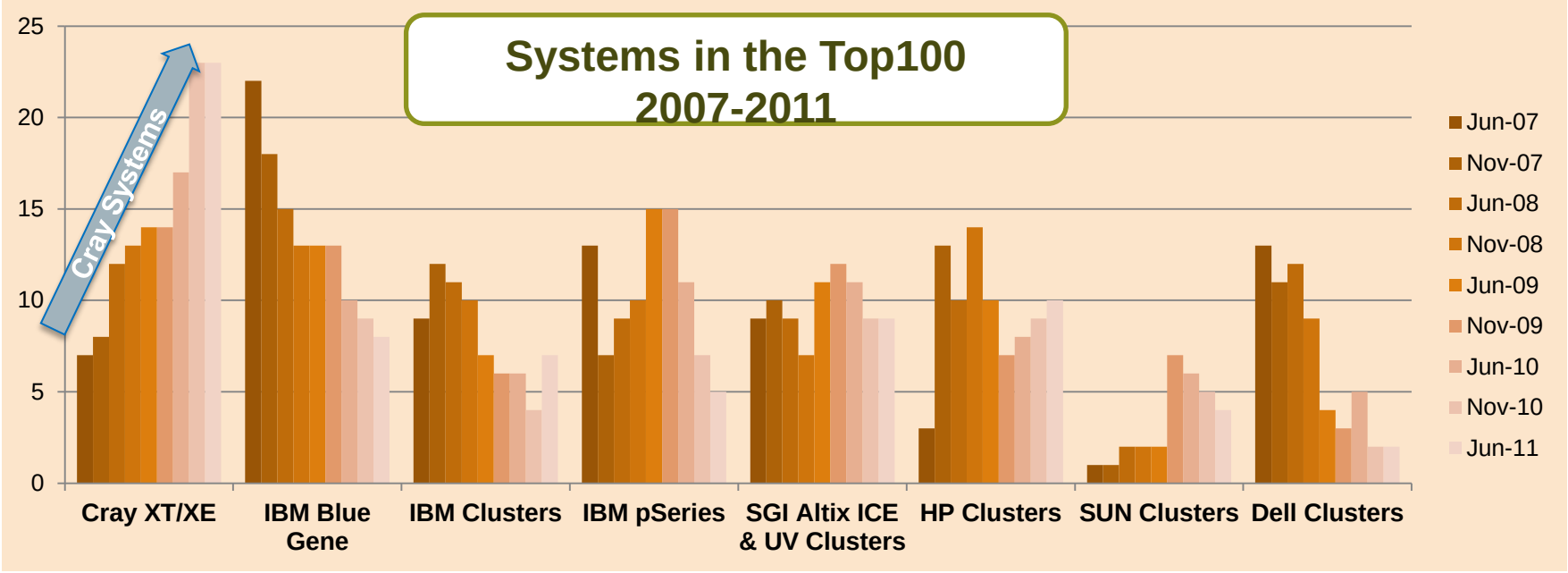
Top 500 Supercomputers in the World

November 2011

TOP500 SUPERCOMPUTER SITES

	Top 10	Top 50	Top 100
Cray Systems	3	15	21
Vendor Rank	#1	#1	#2

Source: www.top500.org



Our Roadmap: Adaptive Supercomputing

2010

Baker (XE6)

2011

Glacier (XK6)

2013

Cascade

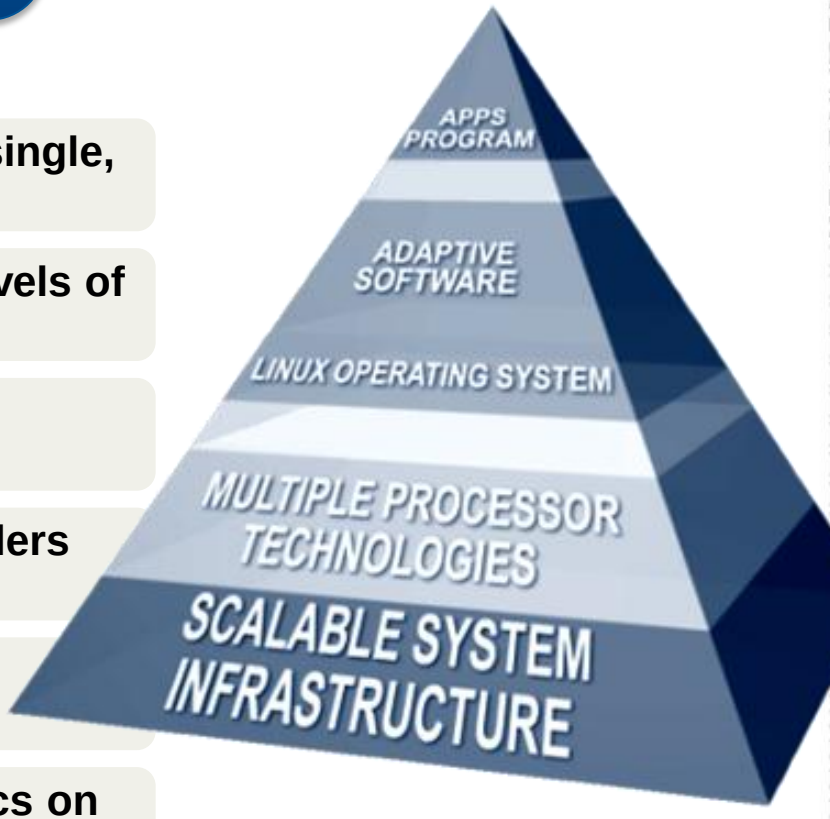
~2016

Next Gen

~2019

Future

- Combine multiple processing technologies into a single, scalable system
- Increasing energy-efficiency and reliability at all levels of the system
- Designing for extreme scale & concurrency
- Focusing on programming tools, libraries & compilers for productivity
- Store and manage data at extreme scale
- Enable science, engineering and advanced analytics on one platform



Outline

- Motivation
- Why a new programming model
- OpenACC overview
- The Cray programming environment for accelerated computing
- Porting and optimization strategies
- 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".

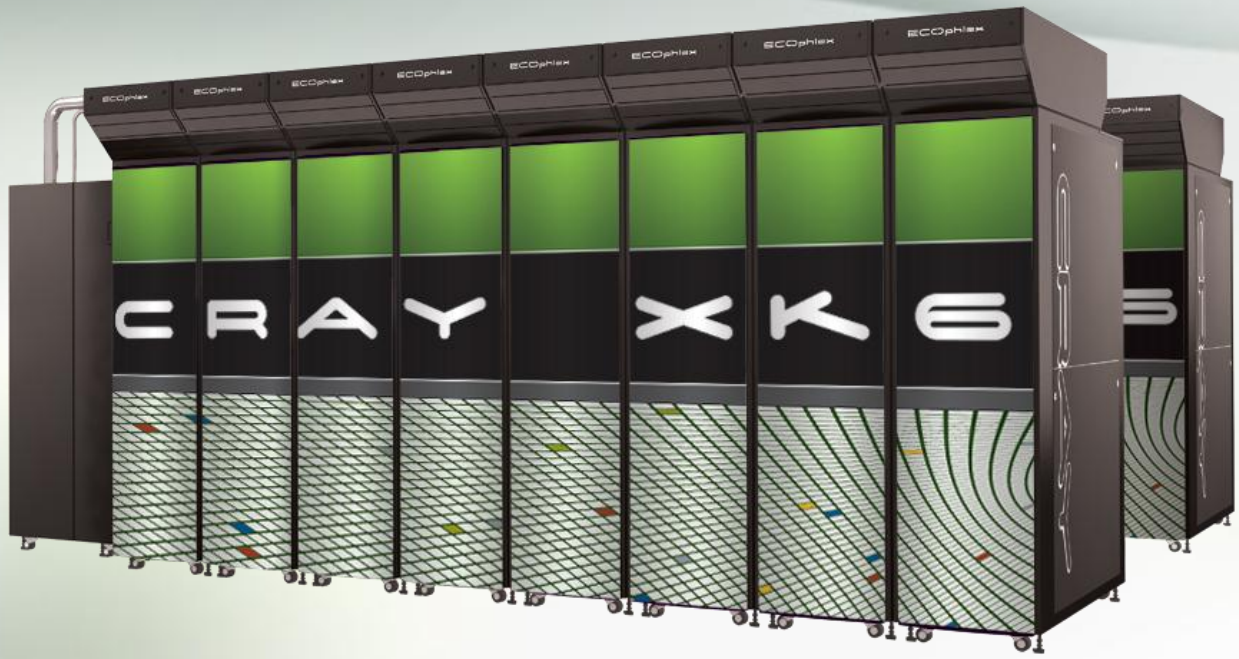
"An exascale supercomputer will take this a step further, connecting tens of thousands of many-core nodes.

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

EPCC News, issue 70 (Autumn 2011)

Three Levels of Parallelism Required

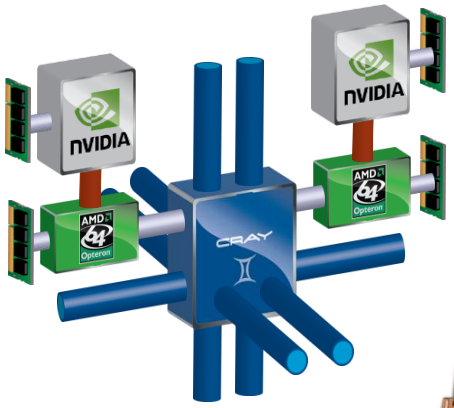
- Developers will continue to use MPI between nodes or sockets
- Developers must address using a shared memory programming paradigm on the node
- Developers must vectorize low level looping structures
- While there is a potential acceptance of new languages for addressing all levels directly. Most developers 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 XK6"
Accelerating the Way to Better Science

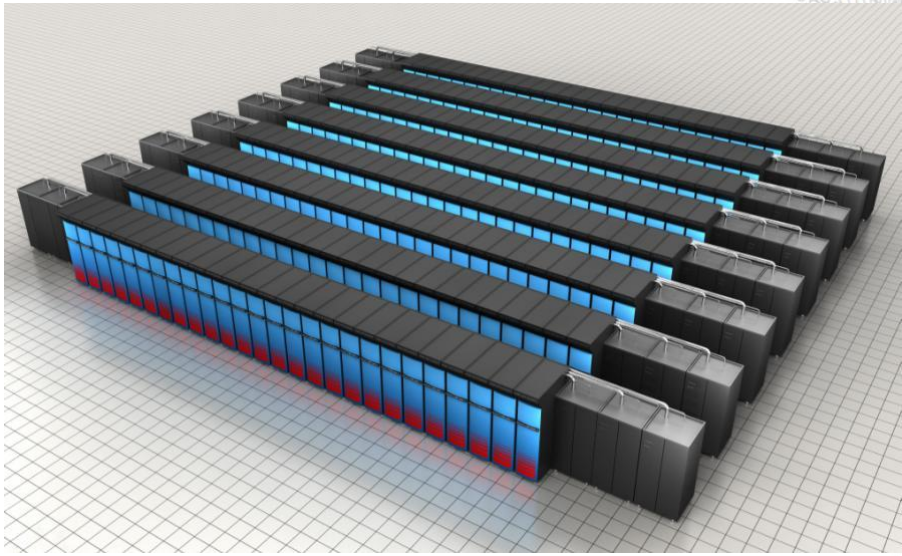
The Cray XK6 hybrid architecture

- Announced in May 2011
- NVIDIA Fermi X2090 GPU
  - Upgradable to Kepler
- AMD Interlagos CPU
- Cray Gemini interconnect
  - high bandwidth/low latency scalability



- Unified X86/GPU programming environment
- Fully compatible with Cray XE6 product line
- Fully upgradeable from Cray XT/XE systems

Upcoming Heterogeneous Multi Petaflop Systems in the US



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

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

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**
    - Non-executable statements (comments, pragmas)
    - 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 80% of the performance obtained with hand coded CUDA
      - Already seeing this in many cases, more tuning ongoing

Motivating Example: Reduction

- Sum elements of an array
- Original Fortran code

```
a=0.0

do i = 1,n
 a = a + b(i)
end do
```

The reduction code in simple CUDA

```
__global__ void reduce0(int *g_idata, int
*g_odata)
{
extern __shared__ int sdata[];

unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x +
threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();

for(unsigned int s=1; s < blockDim.x; s *= 2) {
if ((tid % (2*s)) == 0) {
sdata[tid] += sdata[tid + s];
}
__syncthreads();
}

if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}


extern "C" void reduce0_cuda_(int *n, int *a,
int *b)
{
int *b_d,red;
const int b_size = *n;


cudaMalloc((void **) &b_d , sizeof(int)*b_size);
cudaMemcpy(b_d, b, sizeof(int)*b_size,
cudaMemcpyHostToDevice);
```

```
dim3 dimBlock(128, 1, 1);
dim3 dimGrid(2048, 1, 1);
dim3 small_dimGrid(16, 1, 1);

int smemSize = 128 * sizeof(int);
int *buffer_d, *red_d;
int *small_buffer_d;

cudaMalloc((void **) &buffer_d ,
sizeof(int)*2048);
cudaMalloc((void **) &small_buffer_d ,
sizeof(int)*16);
cudaMalloc((void **) &red_d , sizeof(int));


reduce0<<< dimGrid, dimBlock, smemSize >>>(b_d,
buffer_d);

reduce0<<< small_dimGrid, dimBlock, smemSize
>>>(buffer_d, small_buffer_d);

reduce0<<< 1, 16, smemSize >>>(small_buffer_d,
red_d);

cudaMemcpy(&red, red_d, sizeof(int),
cudaMemcpyDeviceToHost);

*a = red;

cudaFree(buffer_d);
cudaFree(small_buffer_d);
cudaFree(b_d);
}
```

The reduction code in optimized CUDA

```
template<class T>
struct SharedMemory
{
    __device__ inline operator       T*()
    {
        extern __shared__ int __smem[];
        return (T*)__smem;
    }

    __device__ inline operator const T*() const
    {
        extern __shared__ int __smem[];
        return (T*)__smem;
    }
};

template <class T, unsigned int blockSize, bool nIsPow2>
__global__ void
reduce6(T *g_idata, T *g_odata, unsigned int n)
{
    T *sdata = SharedMemory<T>();

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockSize*2 + threadIdx.x;
    unsigned int gridSize = blockSize*2*gridDim.x;

    T mySum = 0;
    while (i < n)
    {
        mySum += g_idata[i];
        if (nIsPow2 || i + blockSize < n)
            mySum += g_idata[i+blockSize];
        i += gridSize;
    }
sdata[tid] = mySum;
    __syncthreads();

    if (blockSize >= 512) { if (tid < 256) { sdata[tid] = mySum = mySum
+ sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] = mySum = mySum
+ sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid <  64) { sdata[tid] = mySum = mySum
+ sdata[tid +  64]; } __syncthreads(); }
```

```
  if (tid < 32)
   {
      volatile T* smem = sdata;
      if (blockSize >=  64) { smem[tid] = mySum = mySum + smem[tid + 32];  }
      if (blockSize >=  32) { smem[tid] = mySum = mySum + smem[tid + 16];  }
      if (blockSize >=  16) { smem[tid] = mySum = mySum + smem[tid + 8];  }
      if (blockSize >=   8) { smem[tid] = mySum = mySum + smem[tid + 4];  }
      if (blockSize >=   4) { smem[tid] = mySum = mySum + smem[tid + 2];  }
      if (blockSize >=   2) { smem[tid] = mySum = mySum + smem[tid + 1];  }
   }

    if (tid == 0)
       g_odata[blockIdx.x] = sdata[0];
}
extern "C" void reduce6_cuda_(int *n, int *a, int *b)
{
   int *b_d;
   const int b_size = *n;

   cudaMalloc((void **) &b_d , sizeof(int)*b_size);
   cudaMemcpy(b_d, b, sizeof(int)*b_size, cudaMemcpyHostToDevice);

   dim3 dimBlock(128, 1, 1);
   dim3 dimGrid(128, 1, 1);
   dim3 small_dimGrid(1, 1, 1);
   int smemSize = 128 * sizeof(int);
   int *buffer_d;
   int small_buffer[4],*small_buffer_d;

   cudaMalloc((void **) &buffer_d , sizeof(int)*128);
   cudaMalloc((void **) &small_buffer_d , sizeof(int));
   reduce6<int,128,false><<< dimGrid, dimBlock, smemSize >>>(b_d,buffer_d,
b_size);
   reduce6<int,128,false><<< small_dimGrid, dimBlock, smemSize
>>>(buffer_d, small_buffer_d,128);
   cudaMemcpy(small_buffer, small_buffer_d, sizeof(int),
cudaMemcpyDeviceToHost);

   *a = *small_buffer;

   cudaFree(buffer_d);
   cudaFree(small_buffer_d);
   cudaFree(b_d);
}
```

The reduction code in OpenACC

- **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

```
!$acc data present(a,b)
!$acc parallel

a = 0.0

!$acc loop reduction(+:a)

do i = 1,n
  a = a + b(i)
end do

!$acc end parallel
!$acc end data
```

Reduction Example Compilation Messages

```
 90.        subroutine sum_of_int_4(n,a,b)
 91.        use global_data
 92.        integer*4 a,b(n)
 93.        integer*8 start_clock, elapsed_clocks, end_clock
 94.        !$acc data present(a,b)
 95.  G----< !$acc parallel
 96.  G      a = 0.0
 97.  G      !$acc loop reduction(+:a)
 98.  G g--< do i = 1,n
 99.  G g      a = a + b(i)
100.  G g--> end do
101.  G----> !$acc end parallel
102.        !$acc end data
103.        end subroutine sum_of_int_4
```

```
ftn-6413 ftn: ACCEL File = gpu_reduce_int_cce.F90, Line = 94
  A data region was created at line 94 and ending at line 107.

ftn-6405 ftn: ACCEL File = gpu_reduce_int_cce.F90, Line = 95
  A region starting at line 95 and ending at line 101 was placed on
the accelerator.

ftn-6430 ftn: ACCEL File = gpu_reduce_int_cce.F90, Line = 98
  A loop starting at line 98 was partitioned across the threadblocks
and the 128 threads within a threadblock.
```

Reduction code summary

Summary of code complexity and performance

Programming Model	Unit of computation	Lines of code	Performance in Gflops (higher is better)	Performance normalized to X86 core
Fortran	Single x86 core	4	2.0 Gflops	1.0
Simple CUDA	GPU	30	1.74 Gflops	0.87
Optimized CUDA	GPU	69	10.5 Gflops	5.25
OpenACC	GPU	9	8.32 Gflops	4.16

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**.
  - Host can usually queue a sequence of operations
    - To be executed on the device, one after the other.

OpenACC Memory model

- **Distinct memory spaces** on the host and device
  - Different locations, different address space
  - Data movement performed by host using runtime library calls that explicitly move data between the separate memories
- GPUs have a **weak memory model**
  - **No synchronisation** between different execution units (SMs)
    - **Unless explicit memory barrier**
  - **One can write OpenACC kernels with race conditions**
    - Giving inconsistent execution results
    - Compiler will catch most errors, but not all (no user-managed barriers)
- OpenACC
  - **Data movement between the memories implicit**
    - **Managed by the compiler**,
    - Based on directives from the programmer.
  - Device memory caches are managed by the compiler
    - With **hints from the programmer** in the form of directives

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)

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

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] (pcopy)
    - 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

OpenACC Async Clause

- **async(handle): like CUDA streams**
  - Allows overlap of tasks on GPU
    - PCIe transfers in both directions
    - Plus multiple kernels (up to 16 with Fermi)
  - **Streams identified by handle**
    - Tasks with same handle execute sequentially
    - can **wait** on one, more or all tasks
      - OpenACC API also allows completeness check
- **First attempt, a simple pipeline:**
  - Processes array, slice by slice
    - Copy data to GPU, process, bring back to CPU
      - Very complicated kernel operation here!
  - Should be able to overlap 3 streams at once
    - Use slice number as stream handle in this case
      - Runtime MODs it back into allowable range
  - Can actually overlap more than three stream
    - No benefit on this test

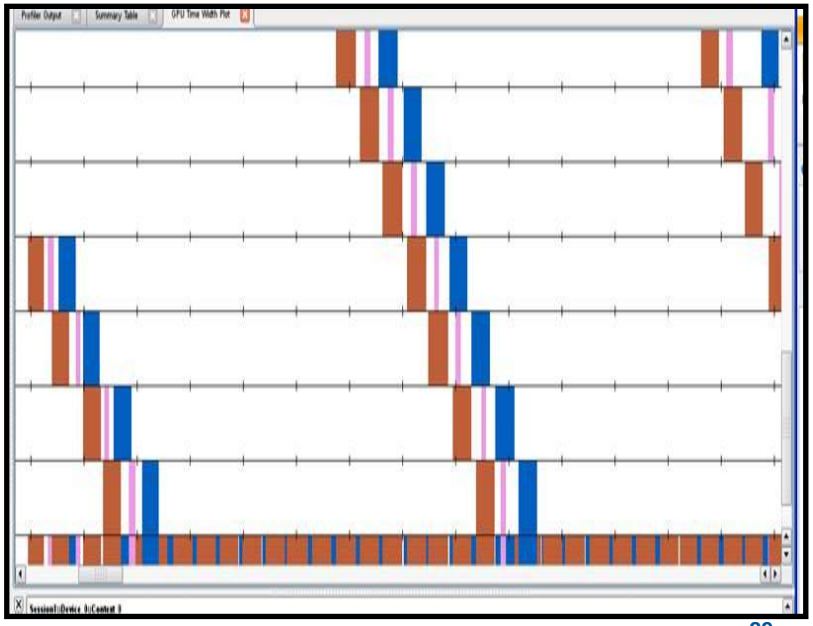
```
INTEGER, PARAMETER :: Nvec = 10000, Nchunks = 10000

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) = SQRT(EXP(a(i,j)*2d0))
    b(i,j) = LOG(b(i,j)**2d0)/2d0
  ENDDO

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

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

OpenACC async Results

- **Execution times (on Cray XK6):**
  - CPU: 3.98s
  - OpenACC, blocking: 3.6s
  - OpenACC, async: 0.82s
  - OpenACC, full async: 0.76s

```
INTEGER, PARAMETER :: Nvec = 10000, Nchunks = 10000

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) = SQRT(EXP(a(i,j)*2d0))
    b(i,j) = LOG(b(i,j)**2d0)/2d0
  ENDDO

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

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

- **NVIDIA Visual profiler:**
  - Time flows to right, streams stacked vertically
    - red: data transfer to GPU
    - pink: computational kernel on GPU
    - blue: data transfer from GPU
  - vertical slice shows what is overlapping
    - only 7 of 16 streams fit in window
    - collapsed view at bottom
  - async handle modded by number of streams
    - so see multiple coloured bars per stream



OpenACC in CCE

- man intro_openacc
- Which module to use, craype-accel-nvidia20
- Forces dynamic linking
- Single object file
- Whole program
- Messages/list file
- Compiles to PTX not CUDA
- Debugger sees original program not CUDA intermediate source

- OpenACC directives status in CCE
  - Only ~~two~~ one constructs are un-implemented
    - Cache
    - ~~Declare~~
  - One unimplemented data clause
    - deviceptr

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

Accelerator Performance Statistics

- **Default statistics collected when accelerated directives are encountered with event tracing**
  - Host time for kernel launches, data copies and synchronization with the accelerator
  - Accelerator time for kernel execution and data copies
  - Data copy size to and from the accelerator
- **Collection enabled by default for programs built with CCE**
- **Collection enabled with runtime environment variable for CUDA**
- **Sampling will not produce accelerator table in the report, but samples can show up in CUDA libraries**

Accelerator Table Column Definitions

- **Host Time%**
  - Percentage of wallclock time for events
- **Host Time**
  - Wallclock time, in seconds, for the event
- **Acc Time**
  - Amount of time the event executed on the accelerator
- **Acc Copy In**
  - Amount of data copied to the accelerator
- **Acc Copy Out**
  - Amount of data copied from the accelerator
- **Calls**
  - The number of time the event occurred

All of the above are summed for regions and functions

Accelerator Statistics

```
Table 2:  Time and Bytes Transferred for Accelerator Regions

  Host  |    Host  |     Acc  | Acc Copy  | Acc Copy  | Calls  |Calltree
 Time%  |    Time  |    Time  |       In  |      Out  |        | PE=HIDE
        |          |          | (MBytes)  | (MBytes)  |        |

 100.0% |   42.787 |   35.429 |  2554.726 |  2559.820 |  38164 |Total
|-------------------------------------------------------------------------------------------
| 100.0% |   42.787 |   35.429 |  2554.726 |  2559.820 |  38164 |himenobmtxp_
|        |          |          |           |           |        | himenobmtxp_.ACC_DATA_REGION@li.65
3  99.6% |   42.628 |   35.273 |  2554.726 |  2559.820 |  38152 |  jacobi_
4        |          |          |           |           |        |   jacobi_.ACC_DATA_REGION@li.227
|||||-------------------------------------------------------------------------------------
5||||  67.4% |   28.836 |   28.168 |     0.004 |     0.004 |   5015 |jacobi_.ACC_REGION@li.309
6||||  66.2% |   28.324 |       -- |        -- |        -- |   1003 | jacobi_.ACC_REGION@li.309(exclusive)
5||||  10.2% |    4.384 |    3.786 |        -- |        -- |   4012 |jacobi_.ACC_REGION@li.334
6||||   9.5% |    4.050 |       -- |        -- |        -- |   1003 | jacobi_.ACC_REGION@li.334(exclusive)
5||||   4.7% |    1.998 |       -- |        -- |        -- |      2 |jacobi_.ACC_DATA_REGION@li.227(exclusive)
5||||   2.6% |    1.113 |    0.513 |        -- |        -- |   4012 |jacobi_.ACC_REGION@li.274
6||||   1.8% |    0.778 |       -- |        -- |        -- |   1003 | jacobi_.ACC_REGION@li.274(exclusive)
```

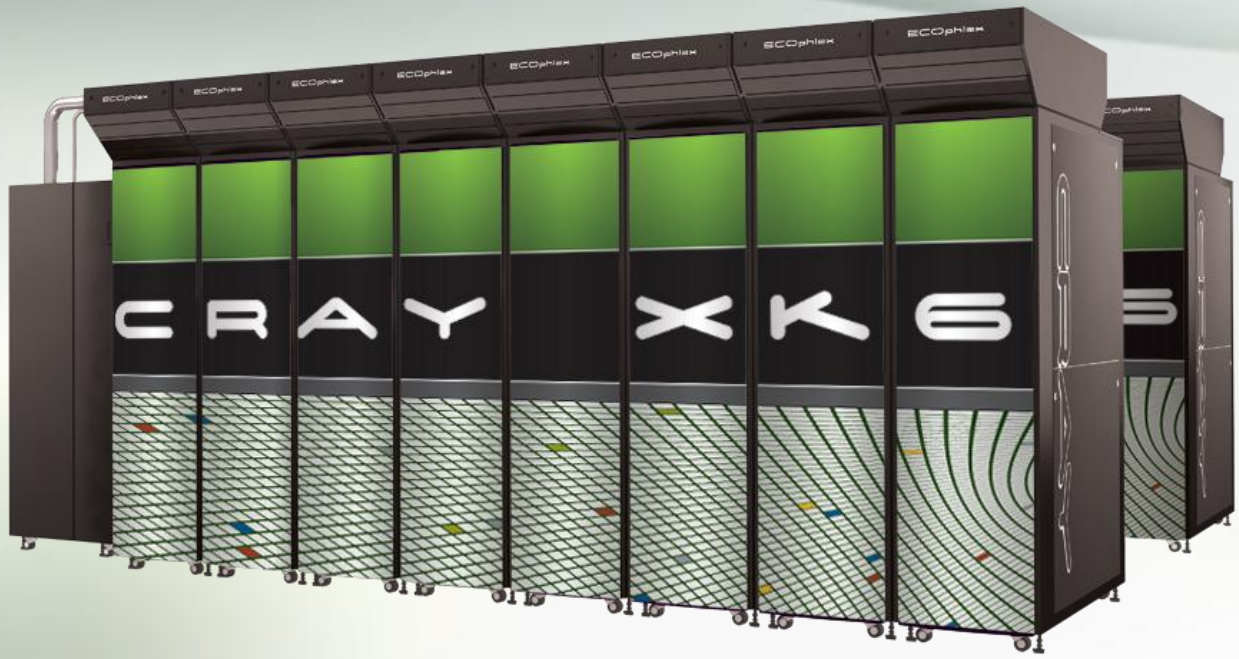
Accelerator Hardware Performance Counters

- **A predefined set of counter groups has been created for ease of use**
  - Combines events that can be counted together
- **ACCPC groups start at 1000, and will be incremented by 100 as new families of accelerators are supported**
- **Specify group by number or name**
  - PAT_RT_ACCPC=1000 *OR*
  - PAT_RT_ACCPC=inst_exec_gst
- **accpc(5) man page provides list of groups and their descriptions**

Accelerator Hardware Counters Statistics

Table 3: ACC Performance Counter Data

```
 warps_launched  | active_warps  | active_cycles  | sm_efficiency  | achieved_occupancy  |Calltree

              16380 |    1976842921 |       61837413 |           2.7% |                66.6% |Total
|-------------------------------------------------------------------------------------------------------------------------------------------------------
|             16380 |    1976842921 |       61837413 |           2.7% |                66.6% |main
|                   |               |                |                |                      | test1
3                   |               |                |                |                       | test1.ACC_REGION@li.35
||||-----------------------------------------------------------------------------------------------------------------------------------------------
4|||            16380 |    1976842921 |       61837413 |           3.1% |                66.6% |test1.ACC_KERNEL@li.35
4|||                0 |             0 |              0 |           0.0% |                   -- |test1.ACC_COPY@li.35
4|||                0 |             0 |              0 |           0.0% |                   -- |test1.ACC_COPY@li.39
|===============================================================================================================
```



Porting and Optimizing for the XK6 System

Cray XK6 vs Cray XE6 Programming

- **Single MPI per node**
  - Multiple processes cannot share the GPU
  - 1 MPI rank per node is likely not what has been used for the CPU code: possibly need to review communication optimization
  - Consider MPI communication/computation overlap
    - Used with core specialization to reserve a core/node for the helper threads
- **Large OpenMP threading**
  - It is very important to get cooperation between CPUs and GPU
  - If there is not enough work for 16 OpenMP threads, try running in single stream mode, with 1 thread per Bulldozer module

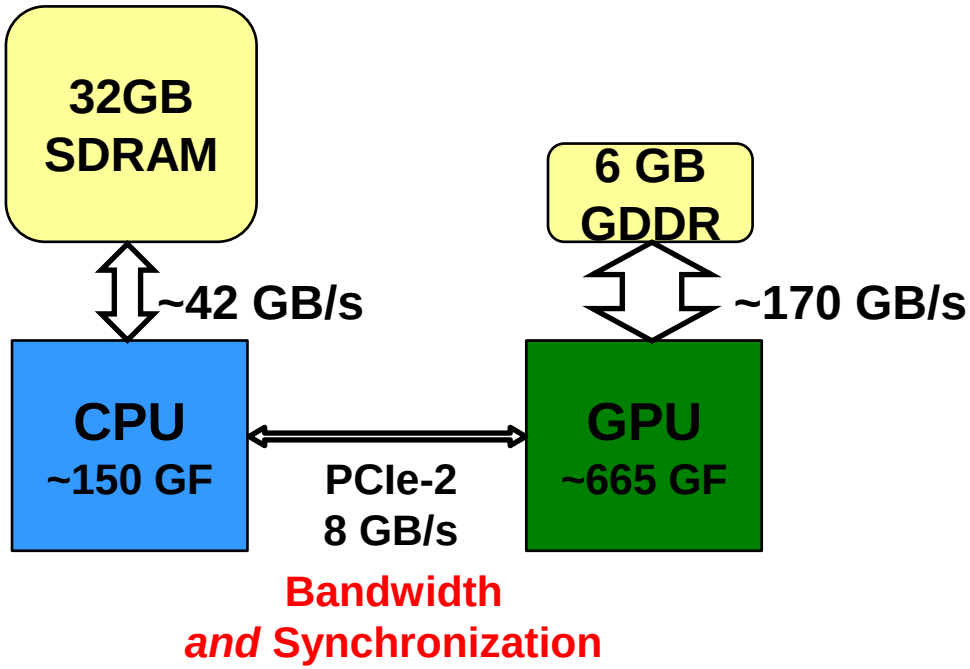
```
export OMP_NUM_THREADS=8
aprun -N 1 -d 8 -cc 0,2,4,6,8,10,12,14 -n XX a.out
```

- **Dynamic linking**
  - You need to be aware that dynamic linking is required with GPU codes
  - Dynamic libraries should be automatically set by the build process
  - In some cases dynamically linked codes can be slower than statically linked ones

A Porting and Optimization Strategy for Multi-core Systems

- Reduce the number of MPI ranks per node
- Add parallelism to MPI ranks to take advantage of cores within a node while **minimizing network injection contention**
- **Maximize on-node communication** between MPI ranks
- **Relieve on-node shared resource contention** by pairing threads or processes that perform different work (for example computation with off-node communication) on the same node
- **Accelerate work intensive parallel loops**

Structural Issues with Accelerated Computing



- **Trick is to keep kernel data structures resident in GPU memory as much as possible**
  - **Avoid copying** between CPU and GPU
  - **Use async, non-blocking, communication, multi-level overlapping**

A three-task approach

- **How to move to a hybrid code**

1. Identification of possible accelerator kernels
    - Determine where to add additional levels of parallelism
        - Assumes MPI application is functioning correctly on X86
        - Find top work-intensive loops (perftools + CCE loop work estimates)

2. Parallel analysis, scoping and vectorization
    - Split loop work among threads
        - Do parallel analysis and restructuring on targeted high level loops
        - Use CCE loopmark feedback, Reveal loopmark and source browsing

3. Moving to OpenMP and then to OpenACC
    - Add parallel directives and acceleration extensions
        - Insert OpenMP directives (Reveal scoping assistance)
        - Run on X86 to verify application and check for performance improvements
        - Convert desired OpenMP directives to OpenACC

Collecting Loop Statistics

- Need to be using CCE

```
module load PrgEnv-cray perftools
```

- Fresh compile AND link with **-h profile_generate**

```
cc -h profile_generate -c my_program.c
cc -h profile_generate -o my_program my_program.o
```

- Instrument binary for event tracing

```
pat_build -u my_program
```
(or `-w` option)

- Run application

- Create report with loop statistics

```
pat_report my_program+pat.xf > loops_report
```

Loop Work Estimates Report

```
Table 3:  Inclusive Loop Time from -hprofile_generate

 Loop Incl  |    Loop  |  Loop  |  Loop  |Function=/.LOOP[.]
      Time  |     Hit  | Trips  | Trips  | PE=HIDE
     Total  |          |   Min  |   Max  |
|-----------------------------------------------------------
|
| 175.676881 |        2 |      0 |   1003 |jacobi_.LOOP.07.li.267
|   0.917107 |     1003 |      0 |    260 |jacobi_.LOOP.08.li.276
|   0.907515 |   129888 |      0 |    260 |jacobi_.LOOP.09.li.277
|   0.446784 |     1003 |      0 |    260 |jacobi_.LOOP.10.li.288
|   0.425763 |   129888 |      0 |    516 |jacobi_.LOOP.11.li.289
|   0.395003 |     1003 |      0 |    260 |jacobi_.LOOP.12.li.300
|   0.374206 |   129888 |      0 |    516 |jacobi_.LOOP.13.li.301
| 126.250610 |     1003 |      0 |    256 |jacobi_.LOOP.14.li.312
| 126.223035 |   127882 |      0 |    256 |jacobi_.LOOP.15.li.313
| 124.298650 | 16305019 |      0 |    512 |jacobi_.LOOP.16.li.314
|  20.875086 |     1003 |      0 |    256 |jacobi_.LOOP.17.li.336
|  20.862715 |   127882 |      0 |    256 |jacobi_.LOOP.18.li.337
|  19.428085 | 16305019 |      0 |    512 |jacobi_.LOOP.19.li.338
|===========================================================================
```

subroutine

internal label

line number

nested loops
- Loop Hits multiply
- Incl Times reduce

A Porting and Optimization Strategy

- **Preparation: add checksum(s) and high-res timer to code**
  - Check for correctness very frequently
  - Profile code on the host
    - Use representative-sized problem, map call tree,
    - Ideally resolve profile by loop nest and measure typical loop iteration counts
- **First get your application working without data regions**
- **Once you have a correct hybrid code**
  - Run on x86 + GPU and get performance feedback
    - perftools profiling analysis
  - Optimize for data locality and copies to the GPU
    - perftools accelerator statistics
  - Optimize kernel(s) on GPU
    - perftools GPU counter statistics
    - perftools Kernel statistics
  - Optimize core performance on CPU
    - Automatic profiling analysis with CPU HW counter threshold feedback

A Porting and Optimization Strategy

- **Optimizing the data movements**
  - Start in subprograms at bottom of call chain
    - Accelerate individual loop nests using parallel regions
      - Concentrate initially on most computationally expensive
    - Add data regions in subprograms
      - Minimize data movements, use **create** clause where possible
      - May need to accelerate insignificant loop nests to avoid data copies
  - Use available feedback to understand data movement
    - Compiler messages: **-ra** for CCE creates *.lst listing files
    - Runtime commentary: **export CRAY_ACC_DEBUG=[1,2,3]** for CCE
    - NVIDIA compute profiler: **export COMPUTE_PROFILE=1**
    - CrayPat performance measurement and analysis tool (Cray PE only)

Performance Tools Example

```
   t1 = gettime()
   stream_counter = 1
   DO j = 1,Nchunks
    my_stream = Streams(stream_counter)



!$acc parallel loop
    DO i = 1,Nvec
     b(i,j) = SQRT(EXP(a(i,j)*2d0))
     b(i,j) = LOG(b(i,j)**2d0)/2d0
    ENDDO
!$acc end parallel loop



     stream_counter = MOD(stream_counter,3) + 1
   ENDDO
!$acc wait
   t2 = gettime()
!$acc end data
```

Performance Tools Example

```
ftn -rad -hnocaf -c -o toaa2.o toaa2.F90
ftn -rad -hnocaf -o toaa2.x toaa2.o
pat_build -w toaa2.x
aprun toaa2.x+pat
 4999899.3359271679
 Time =  88.750109565826278
Experiment data file written:
/lus/scratch/beyerj/openacc/toaa/toaa2.x+pat+10112-43t.xf
Application 1880125 resources: utime ~83s, stime ~7s
pat_report -T toaa2.x+pat+10112-43t.xf
```

Performance Tools Example

```
Table 1:  Profile by Function Group and Function

 Time%  |     Time  | Imb.   |  Imb.   | Calls  |Group
        |           | Time   | Time%   |        | Function

 100.0% | 88.902394 |    -- |     -- | 5003.0 |Total
|-------------------------------------------------------------------------
| 100.0% | 88.902394 |    -- |     -- | 5003.0 |USER
||------------------------------------------------------------------------
||  75.4% | 67.041165 |    -- |     -- | 1000.0 |toaa_.ACC_COPY@li.59
||  24.3% | 21.629574 |    -- |     -- | 1000.0 |toaa_.ACC_COPY@li.65
||   0.2% |  0.155233 |    -- |     -- |    1.0 |toaa_
||   0.0% |  0.037016 |    -- |     -- | 1000.0 |toaa_.ACC_KERNEL@li.59
||   0.0% |  0.032549 |    -- |     -- | 1000.0 |toaa_.ACC_SYNC_WAIT@li.65
||   0.0% |  0.006752 |    -- |     -- | 1000.0 |toaa_.ACC_REGION@li.59
||   0.0% |  0.000074 |    -- |     -- |    1.0 |exit
||   0.0% |  0.000031 |    -- |     -- |    1.0 |toaa_.ACC_SYNC_WAIT@li.79
||========================================================================
|   0.0% |  0.000000 |    -- |     -- |    0.0 |ETC
|=========================================================================
```

Performance Tools Example

```
Table 2:  Time and Bytes Transferred for Accelerator Regions

  Host  |    Host  |     Acc  |  Acc Copy  | Acc Copy  | Calls  |Calltree
 Time%  |    Time  |    Time  |        In  |       Out |        |
        |          |          |  (MBytes)  | (MBytes)  |        |

 100.0% |   88.749 |   88.697 | 152587.891 | 76293.945 |   5001 |Total
|-------------------------------------------------------------------------------------------
| 100.0% |   88.749 |   88.697 | 152587.891 | 76293.945 |   5001 |toaa_
||------------------------------------------------------------------------------------------
|| 100.0% |   88.749 |   88.697 | 152587.891 | 76293.945 |   5000 |toaa_.ACC_REGION@li.59
|||-----------------------------------------------------------------------------------------
3||  75.5% |   67.042 |   67.042 | 152587.891 |        -- |   1000 |toaa_.ACC_COPY@li.59
3||  24.4% |   21.630 |   21.630 |         -- | 76293.945 |   1000 |toaa_.ACC_COPY@li.65
3||   0.0% |    0.037 |    0.026 |         -- |        -- |   1000 |toaa_.ACC_KERNEL@li.59
3||   0.0% |    0.033 |       -- |         -- |        -- |   1000 |toaa_.ACC_SYNC_WAIT@li.65
3||   0.0% |    0.007 |       -- |         -- |        -- |   1000 |toaa_.ACC_REGION@li.59(exclusive)
|||=========================================================================================
||   0.0% |    0.000 |       -- |         -- |        -- |      1 |toaa_.ACC_SYNC_WAIT@li.79
|=============================================================================================
Processing step 3 of 3
```

Performance Tools Example

```
ACC: Transfer 2 items (to acc 1600000000 bytes, to host 0 bytes) from toaa2.F90:55
ACC: Execute kernel toaa_$ck_L55_1 async(auto) from toaa2.F90:55
ACC: Wait async(auto) from toaa2.F90:61
ACC: Transfer 2 items (to acc 0 bytes, to host 800000000 bytes) from toaa2.F90:61
ACC: Transfer 2 items (to acc 1600000000 bytes, to host 0 bytes) from toaa2.F90:55
ACC: Execute kernel toaa_$ck_L55_1 async(auto) from toaa2.F90:55
ACC: Wait async(auto) from toaa2.F90:61
ACC: Transfer 2 items (to acc 0 bytes, to host 800000000 bytes) from toaa2.F90:61
ACC: Transfer 2 items (to acc 1600000000 bytes, to host 0 bytes) from toaa2.F90:55
ACC: Execute kernel toaa_$ck_L55_1 async(auto) from toaa2.F90:55
ACC: Wait async(auto) from toaa2.F90:61
ACC: Transfer 2 items (to acc 0 bytes, to host 800000000 bytes) from toaa2.F90:61
ACC: Transfer 2 items (to acc 1600000000 bytes, to host 0 bytes) from toaa2.F90:55
ACC: Execute kernel toaa_$ck_L55_1 async(auto) from toaa2.F90:55
ACC: Wait async(auto) from toaa2.F90:61
ACC: Transfer 2 items (to acc 0 bytes, to host 800000000 bytes) from toaa2.F90:61
ACC: Transfer 2 items (to acc 1600000000 bytes, to host 0 bytes) from toaa2.F90:55
ACC: Execute kernel toaa_$ck_L55_1 async(auto) from toaa2.F90:55
ACC: Wait async(auto) from toaa2.F90:61
```

Performance Tools Example

```
#ifdef USE_DATA
!$acc data create(a,b)
#endif
  t1 = gettime()
  stream_counter = 1
  DO j = 1,Nchunks
   my_stream = Streams(stream_counter)
#ifdef USE_DATA
!$acc update device(a(:,j))
#endif
!$acc parallel loop
    DO i = 1,Nvec
     b(i,j) = SQRT(EXP(a(i,j)*2d0))
     b(i,j) = LOG(b(i,j)**2d0)/2d0
    ENDDO
!$acc end parallel loop
#ifdef USE_DATA
!$acc update host(b(:,j))
#endif
    stream_counter = MOD(stream_counter,3) + 1
  ENDDO
!$acc wait
  t2 = gettime()
!$acc end data
```

```
ftn -rad -hnocaf -DUSE_DATA -c -o toaa2.o toaa2.F90
ftn -rad -hnocaf -DUSE_DATA -o toaa2.x toaa2.o
pat_build -w toaa2.x
aprun toaa2.x+pat
 50000944.502389029
 Time =  4.1188710090027598
Experiment data file written:
/lus/scratch/beyerj/openacc/toaa/toaa2.x+pat+10178-
43t.xf
Application 1880347 resources: utime ~4s, stime ~2s
pat_report -T toaa2.x+pat+10112-43t.xf
```

Performance Tools Example

```
Table 2:  Time and Bytes Transferred for Accelerator Regions

  Host  |    Host  |     Acc  | Acc Copy  | Acc Copy  | Calls  |Calltree
 Time%  |    Time  |    Time  |       In  |      Out  |        |
        |          |          | (MBytes)  | (MBytes)  |        |

 100.0% |   4.148 |   3.714 |  762.939 |  762.939 |  70005 |Total
|-------------------------------------------------------------------------------------------
| 100.0% |    4.148 |    3.714 |   762.939 |   762.939 |  70005 |toaa_
|        |          |          |           |           |        | toaa_.ACC_DATA_REGION@li.27
|||-----------------------------------------------------------------------------------------
3||  67.3% |    2.792 |    2.487 |        -- |   762.939 |  30000 |toaa_.ACC_UPDATE@li.71
||||---------------------------------------------------------------------------------------
4|||  60.0% |    2.487 |    2.487 |        -- |   762.939 |  10000 |toaa_.ACC_COPY@li.71
4|||   6.9% |    0.286 |       -- |        -- |        -- |  10000 |toaa_.ACC_SYNC_WAIT@li.71
4|||   0.4% |    0.018 |       -- |        -- |        -- |  10000 |toaa_.ACC_UPDATE@li.71(exclusive)
||||=======================================================================================
3||  25.7% |    1.066 |    1.055 |   762.939 |        -- |  20000 |toaa_.ACC_UPDATE@li.52
||||---------------------------------------------------------------------------------------
4|||  25.4% |    1.055 |    1.055 |   762.939 |        -- |  10000 |toaa_.ACC_COPY@li.52
4|||   0.3% |    0.011 |       -- |        -- |        -- |  10000 |toaa_.ACC_UPDATE@li.52(exclusive)
||||=======================================================================================
[[[...]]]
Processing step 3 of 3
```

A Porting and Optimization Strategy (2)

- **Move progressively up call chain, adding data regions**
  - Aim to further reduce data movements
    - No problem nesting data regions: use **present** clause on inner ones
    - May need to port insignificant subprograms to avoid data transfers
    - Use **update** for essential data transfers (e.g. data for halo swaps)
- **Now optimize kernel performance (often trial and error)**
  - Perfect loop nests schedule better than imperfect ones
    - e.g. **remove temporary arrays** by manually inlining (eliminate array b)
    - Or **manually privatize arrays** and break loop nest (make 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
```

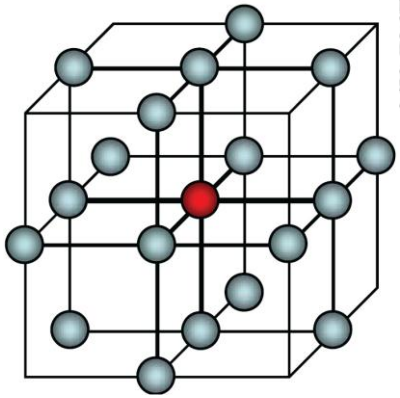
```
DO j = 1,N
 DO i = 0,M+1
  b(i,j) = a(i,j+1) + a(i,j-1)
 ENDDO
ENDDO
DO j = 1,N
 DO i = 1,M
  c(i,j) = b(i+1,j) + b(i-1,j)
 ENDDO
ENDDO
```

A Porting and Optimization Strategy (3)

- **Now look at tweaking the loop scheduling**
  - Quick wins
    - **Optimize loop scheduling**
      - Make sure the right loops are vectorized (for coalesced memory loads)
        - And that they are vectorizable
      - Choose number of workers per gang (threads/block)
        - This number will vary by kernel and by problem size
        - Collapsing or blocking of loops may help (though compilers already do that)
    - See if caching can be used to reduce data loads from device memory
  - Longer term: can loops be migrated up the call chain?
    - e.g. Loop over sites, or blocks of sites (“blocking for cache”)
    - If so, parallelise (gangs) over these

- **Consider overlap of computation and communication using async**
  - **Don’t do this until everything working**
  - May require application restructuring

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 a 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

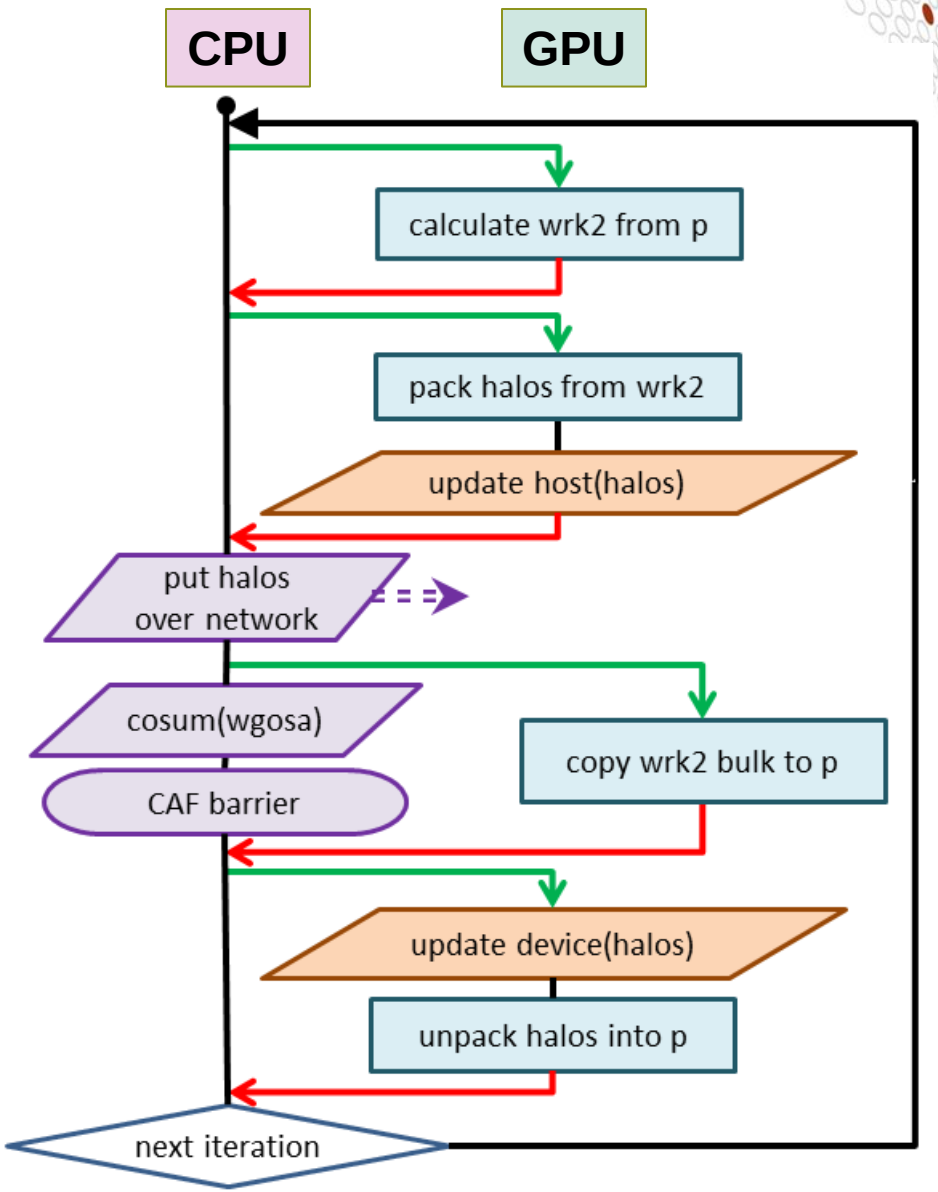
```
DO loop = 1, nn

  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 and 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)**

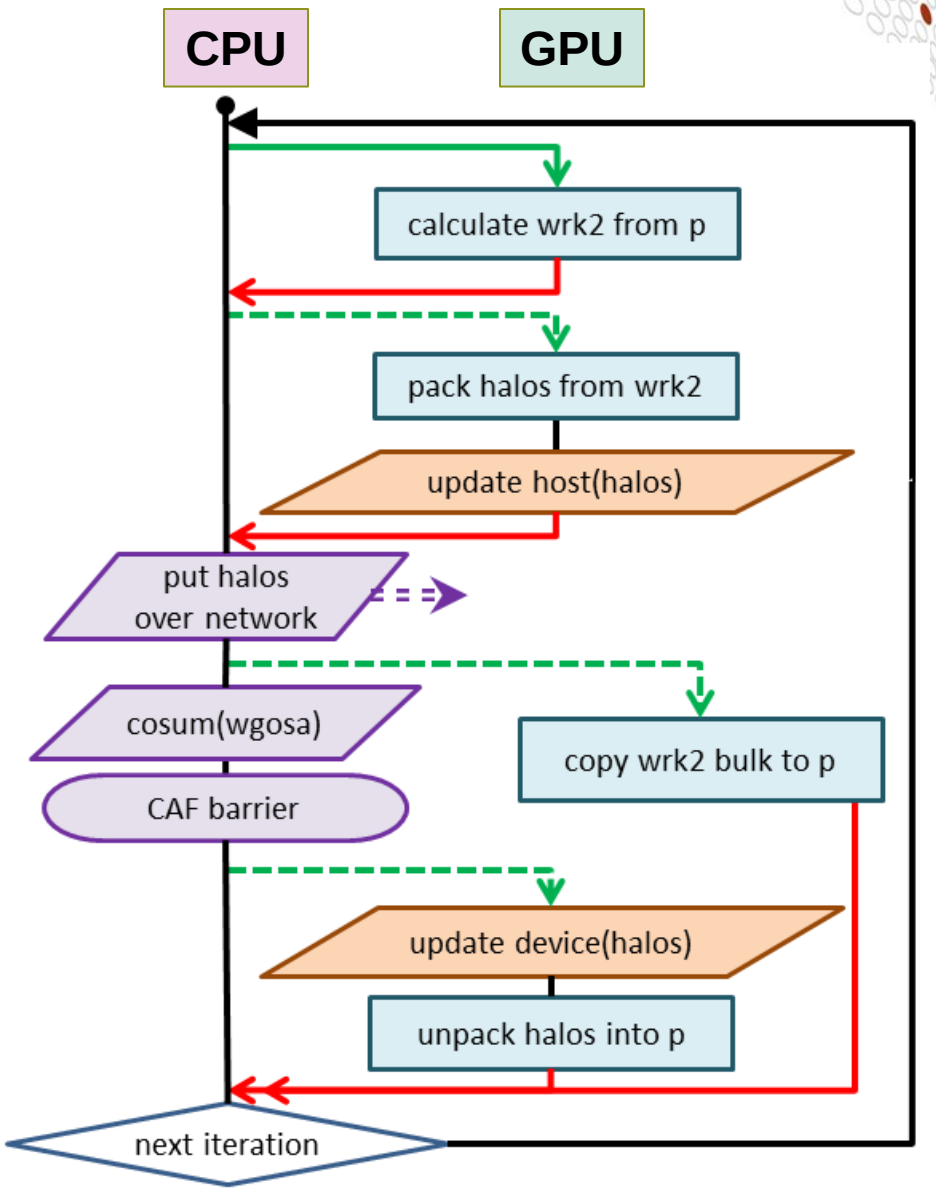
The Himeno GPU code structure

- **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 the **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

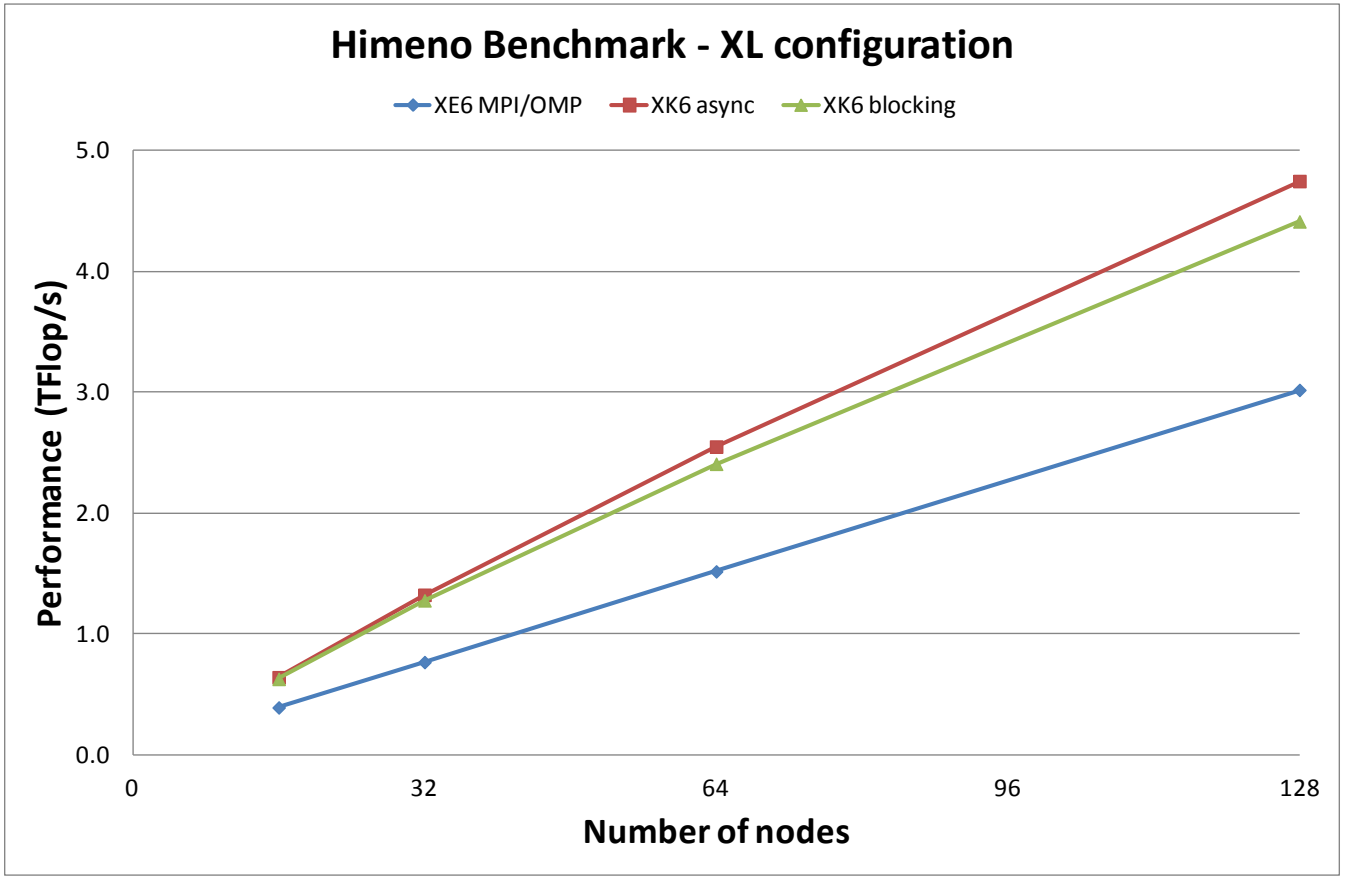
```
!$acc parallel
!$acc loop
DO j = 2,jmax-1
  DO i = 2,imax-1
    sendbuffz_dn(i,j)= wrk2(i,j,2)
    sendbuffz_up(i,j)= wrk2(i,j,kmax-1)
  ENDDO
ENDDO
!$acc end loop
 ...
!$acc loop
 ...
!$acc end loop
!$acc end parallel

!$acc update host
!$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



GAMESS

- **Computational chemistry package suite developed and maintained by the Gordon Group at Iowa State University**
  - http://www.msg.ameslab.gov/gamess/
- **Isolated computationally intensive kernel called CCSD(T)**
  - Method to calculate electronic correlation energy in water clusters
  - ijk-tuples kernel contains iterations of:
    - Communication
    - Complex array transformations
    - Matrix-matrix multiplies
  - Ideal for GPU execution
    - Data movement between host and device can be minimized
    - Kernel is compute intensive with many matrix multiplies
    - Data scrambling can be done on device

OpenACC vs. CUDA

- **Source changes**
  - OpenACC – approximately **75 directives added** to the original source
  - CUDA - **1800 lines of hand-coded** CUDA
- **Performance of ijk-tuples kernel**
  - OpenACC – 36.3 seconds
  - CUDA – 34.8 seconds

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

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

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