Portable and Productive Performance on Hybrid Systems with OpenACC Compilers and Tools

Luiz DeRose
Sr. Principal Engineer
Programming Environments Director
Cray Inc.
Major Hybrid Multi Petaflop Systems in the US

**Blue Waters: Sustained Petascale Performance**
- Production Science at Full Scale
- 244 XE Cabinets + 32 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
The Cray XK7 hybrid architecture

- NVIDIA Kepler (K20) GPUs
- 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
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 asynchronous, non-blocking, communication, multi-level overlapping

Bandwidth and Synchronization

CPU ~150 GF

GPU ~665 GF

32GB SDRAM

6 GB GDDR

~42 GB/s

~170 GB/s

PCIe-2 8 GB/s
Cray’s 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 different 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
Programming for a Node with Accelerator

- Fortran, C, and C++ compilers
  - **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
  - **Debugger support** with DDT and TotalView

- Cray Reveal, built upon an internal compiler database containing a representation of the application
  - Source code browsing tool that provides interface between the user, the compiler, and the performance analysis tool
    - **Scoping tool** to help users port and optimize applications
    - **Performance measurement and analysis** information for identification of main loops of the code to focus refactoring

- **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
    - 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 hybrid 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
    - PGI accelerator directives, CAPS HMPP
      - First steps in the right direction – Needed standardization
    - Possible performance sacrifice
      - A small performance gap is acceptable (do you still hand-code in assembly?)
      - Goal is to provide at least 80% of the performance obtained with hand coded CUDA
  - Compiler support: all complete in 2012
    - Cray CCE: complete in the 8.1 release
    - PGI Accelerator version 12.6 onwards
    - CAPS Full support in version 1.3
Motivating Example: Reduction

- Sum elements of an array
- Original Fortran code
- 2.0 GFlops

```fortran
a = 0.0

do i = 1,n
    a = a + b(i)
end do
```
The reduction code in simple CUDA

```c
__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);

    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, small_buffer_d, sizeof(int), cudaMemcpyDeviceToHost);
    *a = red;

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

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, small_buffer_d, sizeof(int), cudaMemcpyDeviceToHost);
*a = red;
CudaFree(buffer_d);
CudaFree(small_buffer_d);
CudaFree(b_d);
```

1.74 GFlops
The reduction code in optimized CUDA

```cpp
#include "Cray")

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>
struct sdata {
    int mSize;
    T* g_odata;
    T* g_idata;
    T* smem;
};

template <class T, unsigned int blockSize, bool nlsPow2>
__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 (nlsPow2 || 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;
    cudaMemcpy(void **) &buffer_d , sizeof(int)*128);
    cudaMemcpy(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
  - 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

- **8.32 GFlops**
OpenACC Execution Model

- **In short**: It's just like CUDA

- **Host-directed execution** with attached GPU accelerator
- **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**

- **In short: it's just like CUDA**

- **Memory spaces on the host and device, maybe, distinct**
  - Different locations, different address space
  - Data movement performed by host using runtime library calls that explicitly move data between the separate spaces

- **GPUs have a weak memory model**
  - **No synchronization** 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
  - Caching (explicitly use GPU shared memory for reused data)
    - Automatic caching (e.g. NVIDIA Fermi, Kepler)

- User can tune default behavior with optional directives and clauses
- Loop schedule: spreading loop iterations over PEs of GPU
  - Compiler takes care of cases where iterations doesn’t divide threadblock size

<table>
<thead>
<tr>
<th>Parallelism</th>
<th>NVIDIA GPU</th>
<th>SMT node (CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gang:</td>
<td>a threadblock</td>
<td>CPU</td>
</tr>
<tr>
<td>worker:</td>
<td>warp (32 threads)</td>
<td>CPU core</td>
</tr>
<tr>
<td>vector:</td>
<td>SIMT group of threads</td>
<td>SIMD instructions (SSE, AVX)</td>
</tr>
</tbody>
</table>
A First OpenACC Program

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

- 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

- One of the kernels now in subroutine (maybe in separate file)
  - Compiler 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

```plaintext
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
```
CUDA Interoperability

- **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

```plaintext
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 = \n    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();
}
```
OpenACC async Clause

- `async[(handle)]` clause for parallel, update directives
  - Launch accelerator region/data transfer asynchronously
  - Operations with same handle guaranteed to execute sequentially
    - as for CUDA streams
  - Operations with different handles can overlap
    - if the hardware permits it and runtime chooses to schedule it:
      - can potentially overlap:
        - PCIe transfers in both directions
        - Plus multiple kernels
  - can overlap up to 16 parallel streams with Fermi
  - streams identified by handle (integer-valued)
    - tasks with same handle execute sequentially
    - can wait on one or all tasks

- `!$acc wait`: waits for completion of all streams of tasks
  - `!$acc wait(handle)` waits for a specified stream to complete

- Runtime API library functions
  - can also be used to wait or test for completion
OpenACC async Clause

First attempt
- a simple pipeline:
  - processes array, slice by slice
    - copy data to GPU,
    - process on GPU,
    - bring back to CPU
- can overlap 3 streams at once
  - use slice number as stream handle
    - don't worry if number gets too large
    - OpenACC runtime maps it back into allowable range (using MOD function)

```fortran
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)
  !$acc parallel loop async(j)
    DO i = 1,Nvec
      b(i,j) = <function of a(i,j)>
    ENDDO
  !$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

- **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

```fortran
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
```