Cray Programming Environment Update

Luiz DeRose & Heidi Poxon

Cray Inc.

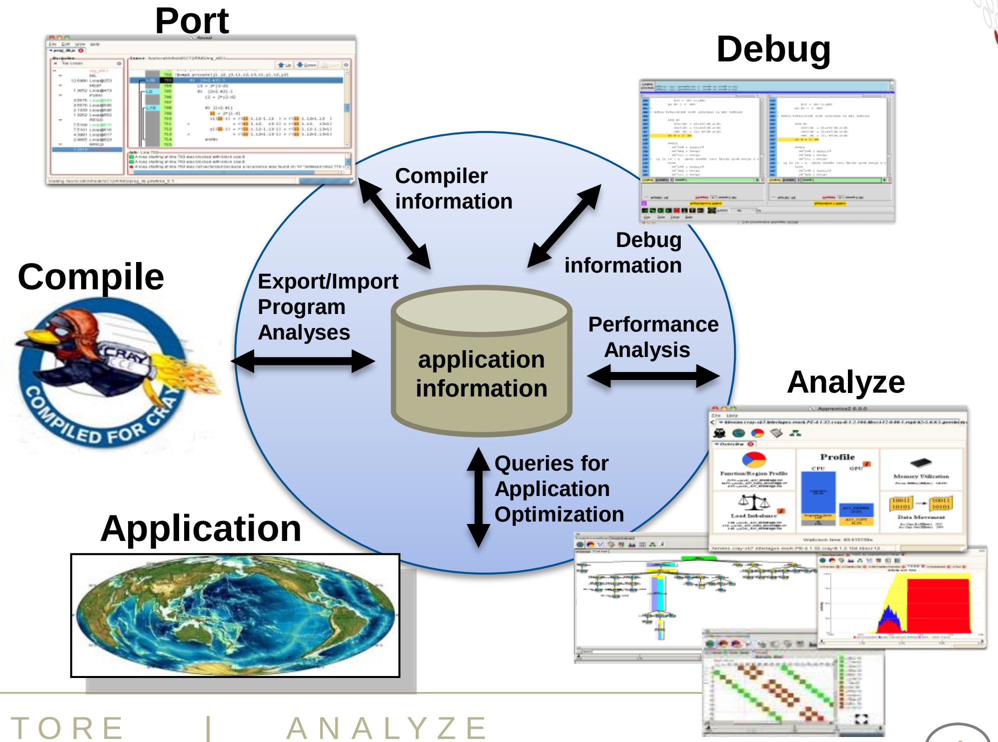
Legal Disclaimer

Schedule (times are guidelines)

09:30 – 09:45	Introductions and Goals
09:45 – 10:00	Cray Programming Environment overview
10:00 – 10:30	CCE Overview and recent enhancements
10:30 – 10:45	Break
10:45 – 11:45	OpenACC and OpenMP 4
11:45 – 12:00	Recent MPI enhancements
12:00 – 13:45	Lunch
13:00 – 13:45	CrayPat overview and recent enhancements
13:45 – 14:30	Using Reveal to add OpenMP
14:30 – 14:45	Break
14:45 – 15:00	Overview of libsci / libsci_acc
15:00 – 15:15	Where to find help
15:15 – 15:30	PE Roadmap
15:30 – 16:00	Questions / Recap
16:00	Adjourn

The Programming Environment Mission

- Focus on **Performance** and **Programmability**
  - It is the role of the Programming Environment to **close the gap** between observed performance and achievable performance
- Support the **application development life cycle** by providing a **tightly coupled** environment with compilers, libraries, and tools that will **hide the complexity** of the system
  - Address issues of scale and complexity of HPC systems
  - Target **ease of use** with extended **functionality** and increased **automation**
  - Close **interaction with users**
    - ➢ For feedback targeting functionality enhancements



Cray Programming Environment

Programming Languages

- Fortran
- C
- C++
- Chapel

Programming models

Distributed Memory (Cray MPT)
- MPI
- SHMEM
- GA

Shared Memory
- OpenMP 4.0
- OpenACC 2.0

PGAS
- UPC
- Fortran Coarrays
- CoArray C++

Compilers

- Cray Compiling Environment (CCE)
- GNU
- 3rd party compiler

Environment setup

- Modules

Tools

Debuggers

- TotalView
- DDT
- lgdb

Debugging Tools

- ATP
- STAT

Performance Analysis

- CrayPat & Cray Apprentice2

Porting Tools

- Reveal
- CCDB

Optimized Scientific Libraries

Dense

- BLAS
- LAPACK
- ScaLAPACK
- Iterative Refinement Toolkit

Sparse

- Cray PETSc (with CASK)
- Cray Trilinos (with CASK)

FFT

- FFTW

I/O Libraries

- NetCDF
- HDF5

Cray developed
Licensed ISV SW
3rd party packaging
Cray added value to 3rd party

The Cray Compiling Environment

- **Cray technology focused on scientific applications**
  - Takes advantage of **automatic vectorization**
  - Takes advantage of **automatic shared memory parallelization**
- **Automatic optimizations for Cray architectures to deliver performance of a new target through simple recompile**
  - Hide system complexity
- **PGAS languages (UPC & Fortran Coarrays) fully optimized and integrated into the compiler**
  - No preprocessor involved
  - Target the network appropriately
  - Full debugger support with Allinea's DDT
- **Focus on standards for application portability and investment protection**
  - Fortran 2008 standard compliant
  - C++11 compliant (working on C++14)
  - OpenMP 4.0 compliant (working on OpenMP 4.5)
  - OpenACC 2.0
  - UPC 1.3

The Cray Compiling Environment

- **Cray technology focused on scientific applications**
  - Takes advantage of **automatic vectorization**
  - Takes advantage of **automatic shared memory parallelization**
- **Automatic optimizations for Cray architectures to deliver performance of a new target through simple recompile**
  - Hide system complexity
- **PGAS languages (UPC & Fortran Coarrays) fully optimized and integrated into the compiler**
  - No preprocessor involved
  - Target the network appropriately
  - Full debugger support with Allinea's DDT
- **Focus on standards for application portability and investment protection**
  - Fortran 2008 standard compliant
  - C++11 compliant (working on C++14)
  - OpenMP 4.0 compliant (working on OpenMP 4.5)
  - OpenACC 2.0
  - UPC 1.3

Cray MPI & Cray SHMEM

- **MPI**
  - Implementation based on MPICH3 from ANL
    - ANL does base MPI standard support, we add new functionality, improve performance both on-node, and all ranges of scale including at very high scale
  - Full MPI-3 support with the exception of
    - MPI-2 Dynamic process management (MPI_Comm_spawn)
  - MPI Forum active participant
  - Participated in the MPICH ABI Consortium
    - ANL MPICH, Intel MPI, IBM PE MPI and Cray MPI

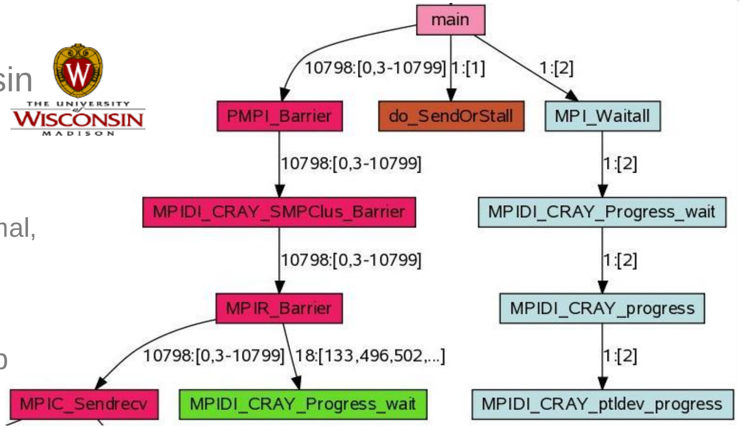
- **Cray SHMEM**
  - Fully optimized Cray SHMEM library supported
    - Cray implementation close to the T3E model
    - Cray XE & XC implementation on top of the Distributed Memory Applications API (DMAPP)

Cray Performance Analysis Tools

- **From performance measurement to performance analysis**
- **Assist the user with application performance analysis and optimization**
  - Help user identify important and meaningful information from potentially massive data sets
  - Help user identify problem areas instead of just reporting data
  - Bring optimization knowledge to a wider set of users
- **Focus on ease of use and intuitive user interfaces**
  - Automatic program instrumentation
  - Automatic analysis
- **Target scalability issues in all areas of tool development**

Debugging on Cray Systems

- **Systems with thousands of threads of execution need a new debugging paradigm**
- **Cray's focus is to build tools around traditional debuggers with innovative techniques for productivity and scalability**
  - **Scalable** Solutions based on MRNet from University of Wisconsin
    - **STAT - Stack Trace Analysis Tool**
      - Scalable generation of a single, merged, stack backtrace tree
    - **ATP - Abnormal Termination Processing**
      - Scalable analysis of a sick application, delivering a STAT tree and a minimal, comprehensive, core file set.
  - LGDB / CCDB
    - Ability to see data from multiple processors in the same instance of lgdb
      - without the need for multiple windows
    - **Comparative debugging**
      - A **data-centric paradigm** instead of the traditional control-centric paradigm
      - Collaboration with University of Queensland
- **Support for traditional debugging mechanism**
  - RogueWave TotalView and Allinea DDT



Cray Adaptive Scientific Libraries

- **The goal of the Cray Scientific Libraries is to provide the Cray user maximum performance with minimum effort**
- **Scientific Libraries today have three concentrations to increase productivity with enhanced performance**
  - **Standardization**
  - **Autotuning**
  - **Adaptive Libraries**
- **Cray adaptive model**
  - Runtime analysis allows **best** library/kernel to be **used dynamically**
  - Extensive offline testing allows **library to make decisions** or remove the need for those decisions
  - Decision depends on the system, on previous performance info, obtained previously, and characteristics of calling problem
- **What makes Cray libraries special:**
  - **Node performance**
  - **Network performance**
  - **Highly adaptive software**

The Cray Compiling Environment

CCE

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

CCE Highlights

- **Arguably the most complete vectorization capabilities in the industry**
  - **Fully automatic loop vectorization** without the need of directives and source code modification
    - This includes **automatic outer loop vectorization**, which is unique in the industry
- **Focus on real applications, instead of just benchmarks**
- **Compiler feedback with annotated listing of source code indicating important optimizations**
- **The Program Library (PL), an application wide repository**
  - Allows **whole application analysis**
  - Allows exchange of information between tools and the compiler
- **Automatic shared memory parallelization with whole program analysis**
- **Bit reproducibility while maintaining high performance is a key example; critical for our climate modeling customers**
- **Fully integrated heterogeneous optimization capability**

CCE flex_mp Support (Bit Reproducibility)

- **Background:**
  - Required by some applications
  - Given a single executable for the application, demonstrate identical floating point results while:
    - Using the same data set
    - Changing the number of MPI ranks
    - Changing the number of OpenMP threads

- **The Cray approach directly addresses the sources of divergence**
  - CCE does **NOT** perform extended precision arithmetic hoping conversion truncates divergence

- **CCE provides the –hflex_mp option for controlling floating point and complex consistency issues related to multiprocessing**
  - -hfp is for floating point and complex consistency issues within a single thread
  - The performance impact depends mostly on how much the performance depends on vectorization of floating point addition and multiplication

- **-hflex_mp=intolerant is the option most certain to provide bit reproducibility, although it also has the highest impact on performance**

- **-hflex_mp=conservative has comparatively little impact on performance, but is not strict enough for some applications'**

Some Cray Compilation Environment Basics

- **CCE-specific features:**
  - Optimization: **-O2** is the default and you should usually use this
  - OpenACC is supported by default if GPU targeting module (craype-accel-nvidia*) is loaded
  - CCE only gives minimal information to stderr when compiling
    - To see more information, you should request a compiler listing file
      - flags **-ra** for ftn or **-hlist=a** for cc
      - writes a file with extension .lst
      - contains annotated source listing, followed by explanatory messages
    - Each message is tagged with an identifier, e.g.: **ftn-6430**
      - to get more information on this, type: **explain <identifier>**
    - Cray Reveal can display all this information (and more)

Recommended CCE Compilation Options

- **Use default optimization levels**
  - It's the equivalent of most other compilers –O3 or –fast
  - It is also our most thoroughly tested configuration

- **Using –O3,fp3 (or –O3 –hfp3, or some variation)**
  - -O3 only gives you slightly more than –O2
  - We also test this thoroughly
  - -hfp3 gives you a lot more floating point optimization, esp. 32-bit
  - **Do not use –Oipa5, -Oaggress, and so on**
    - higher numbers are not always correlated with better performance

- **Optimizing OpenACC**
  - Try –hacc_model=fast_addr
    - This uses 32-bit integers in all addressing to improve GPU performance
    - In rare cases may result in incorrect code

- **Optimizing for compile time rather than execution time**
  - Compile time can sometimes be improved by disabling certain features/optimizations
    - Some common things to try: -hnodwarf, -hipa0, -hunroll0

OpenMP

- **OpenMP is ON by default**
  - Optimizations controlled by **–hthread#**
- **Autothreading is NOT on by default;**
  - -hautothread to turn on
  - Modernized version of Cray X1 streaming capability
  - **Interacts with OpenMP directives**
- **If you do not want to use OpenMP and have OMP directives in the code, make sure to shut off OpenMP at compile time**
  - **To shut off** use **–hthread0** or **–xomp** or **–hnoomp**

Production Quality

- **Functional regression testing done nightly**
  - Roughly 35,000 nightly regression tests run for Fortran (14,000), C (7,000), and C++ (14,000)
  - Default optimization, but for multiple targets (X86, X86+AVX+FMA, X2, X86+NVIDIA), plus “debug” and “production” compiler versions
  - Additionally, cycle through “options testing” with the same test base
    - Fortran: -G0, -G1, -G2, -O0, -Oipa0, -Oipa5 -hpic, “-O3,fp3” –e0
    - C and C++: -Gn, -O0, -hipa0, -hipa5, -hpic, “-O3 –hfp3” -hzero
    - Additional tests and suites have been added for GPU testing
    - And some “stress test” option sets to create worse-case scenarios for the compiler
    - Other combinations as necessary and by request

- **Performance regression testing done weekly using important applications and benchmarks**

- **Automated tools quickly isolate a test change to a specific compiler or library mod.**

Loopmark: Compiler Feedback

- **Compiler can generate an filename.lst file.**
  - Contains annotated listing of your source code with letter indicating important optimizations

```
%%%    L o o p m a r k   L e g e n d    %%%
Primary Loop Type        Modifiers
------- ---- ----        ---------
                         a - vector atomic memory operation
A   - Pattern matched    b - blocked
C   - Collapsed          f - fused
D   - Deleted            i - interchanged
E   - Cloned             m - streamed but not partitioned
I   - Inlined            p - conditional, partial and/or computed
M   - Multithreaded      r - unrolled
P   - Parallel/Tasked    s - shortloop
V   - Vectorized         t - array syntax temp used
W   - Unwound            w - unwound
```

Example: Cray loopmark Messages

- **ftn –rm …** or **cc –hlist=m …**

```
29.   b-------<  do i3=2,n3-1
30.   b b-----<     do i2=2,n2-1
31.   b b Vr--<        do i1=1,n1
32.   b b Vr            u1(i1) = u(i1,i2-1,i3) + u(i1,i2+1,i3)
33.   b b Vr     *        + u(i1,i2,i3-1) + u(i1,i2,i3+1)
34.   b b Vr            u2(i1) = u(i1,i2-1,i3-1) + u(i1,i2+1,i3-1)
35.   b b Vr     *        + u(i1,i2-1,i3+1) + u(i1,i2+1,i3+1)
36.   b b Vr-->        enddo
37.   b b Vr--<        do i1=2,n1-1
38.   b b Vr            r(i1,i2,i3) = v(i1,i2,i3)
39.   b b Vr     *       - a(0) * u(i1,i2,i3)
40.   b b Vr     *       - a(2) * ( u2(i1) + u1(i1-1) + u1(i1+1) )
41.   b b Vr     *       - a(3) * ( u2(i1-1) + u2(i1+1) )
42.   b b Vr-->        enddo
43.   b b----->     enddo
44.   b------->  enddo
```

Example: Cray loopmark messages (cont)

```
 ftn-6289 ftn: VECTOR File = resid.f, Line = 29
  A loop starting at line 29 was not vectorized because a recurrence was found on "U1" between lines
32 and 38.
ftn-6049 ftn: SCALAR File = resid.f, Line = 29
  A loop starting at line 29 was blocked with block size 4.
ftn-6289 ftn: VECTOR File = resid.f, Line = 30
  A loop starting at line 30 was not vectorized because a recurrence was found on "U1" between lines
32 and 38.
ftn-6049 ftn: SCALAR File = resid.f, Line = 30
  A loop starting at line 30 was blocked with block size 4.
ftn-6005 ftn: SCALAR File = resid.f, Line = 31
  A loop starting at line 31 was unrolled 4 times.
ftn-6204 ftn: VECTOR File = resid.f, Line = 31
  A loop starting at line 31 was vectorized.
ftn-6005 ftn: SCALAR File = resid.f, Line = 37
  A loop starting at line 37 was unrolled 4 times.
ftn-6204 ftn: VECTOR File = resid.f, Line = 37
  A loop starting at line 37 was vectorized.
```

Example of Explain Utility

```
users/ldr> explain  ftn-6289

VECTOR:  A loop starting at line %s was not vectorized because a recurrence

was found on "var" between lines num and num.

Scalar code was generated for the loop because it contains a linear
recurrence.  The following loop would cause this message to be issued:

        DO I = 2,100
           B(I) = A(I-1)
           A(I) = B(I)
        ENDDO
```

CCE 8.4 Highlights

- **Support for the C++11 language standard**
  - To enable C++11 features, use the -h std=c++11 command line option
- **Support for the OpenMP 4.0 specification**
- **Support for the inline assembly ASM construct for x86 processor targets**
- **Support for GNU extensions by default (-h gnu option)**
- **Fortran option to initialize floating point arrays to NaNs**

Portable and Productive Performance on Hybrid Systems

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

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 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

Programming for a Node with Accelerator

- **Fortran, C, and C++ compilers**
  - **OpenMP 4.0 Device 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 allinea www.allinea.com DDT, ROGUE WAVE SOFTWARE TotalView, or **Cray CCDB**

- **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)

Accelerator Programming

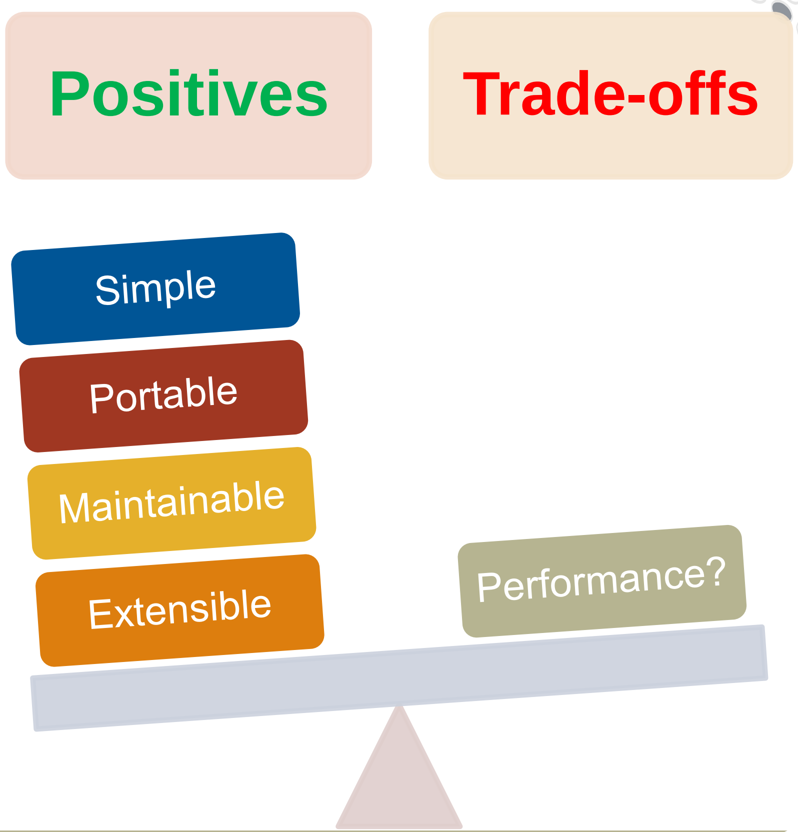
- Why do we need a new GPU programming model?

- Aren't there enough ways to drive a GPU already?
  - CUDA (incl. NVIDIA CUDA-C & PGI CUDA-Fortran)
  - OpenCL

- All are quite low-level and closely coupled to the GPU
  - User needs to rewrite kernels in specialist language:
    - **Hard to write and debug**
    - **Hard to optimise for specific GPU**
    - **Hard to port to new accelerator**
  - Multiple versions of kernels in codebase
    - **Hard to add new functionality**

Accelerator Programming with Directives

- 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
  - Possible performance sacrifice
    - A small performance gap is acceptable (do you still hand-code in assembly?)
    - Cray goal is to provide at least 80% of the performance obtained with hand coded CUDA



Motivating Example: Reduction

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

```
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, *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);
}
```

1.74 GFlops

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), dimGrid(128, 1, 1), 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);
}
```

10.5 GFlops

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
    - Uses MIMD and SIMD parallelism
  - Data movement
    - allocates/frees GPU memory at start/end of region
    - moves data to/from GPU

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

a = 0.0

!$acc update device(a)

!$acc parallel

!$acc loop reduction(+:a)

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

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

8.32 GFlops

The reduction code in OpenMP 4.0

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

```
! Assume outer data region has
! placed a,b on accelerator
a = 0.0

!$omp target update to(a)

!$omp target teams distribute &
!$omp   reduction(+:a)

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

!$omp end target teams distribute
```

OpenMP (and OpenACC) Execution Model

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

- **Host-directed execution with attached accelerator(s)**
- **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)
- **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

OpenMP (and OpenACC) device Memory Model

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

- **Memory spaces on the host and device distinct (usually)**
  - 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**
  - There is **no automatic synchronization** between different execution units (SMs)
    - Unless explicit memory barrier
  - **One can write device kernels with race conditions**
    - Giving inconsistent execution results
    - Compiler will catch most errors, but not all (no user-managed barriers)
- **OpenMP device constructs**
  - **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 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

```
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

Data Clauses

- **Applied to: data, parallel [loop], kernels [loop]**
  - **copy, copyin, copyout**
    - Copy moves data "in" to GPU at start of region and/or "out" to CPU at end
    - Supply list of arrays or array sections (using ":" notation)
    - Fortran uses start:end; C/C++ uses start:length
      - e.g. first N elements: Fortran 1:N (familiar); C/C++ 0:N (less familiar)
      - Advice: be careful and don't make mistakes! ☺
      - Use profiler and/or runtime commentary to see how much data moved
      - Avoid non-contiguous array slices for performance
  - create
    - No copyin/out – useful for shared temporary arrays in loop nests
  - private, firstprivate: as per OpenMP
    - scalars private by default (not just loop variables)
      - Advice: declare them anyway, for clarity

More Data Clauses

- **present, present_or_copy*, present_or_create**
  - pcopy*, pcreate for short
  - Checks if data is already on the device
    - if it is, it uses that version
      - no data copying will be carried out for that data
    - if not, it does the prescribed data copying
- **The data is processed on the GPU**

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)
  - 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**

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

Clauses for !$acc parallel loop

- **Tuning clauses:**
  - User can tune default behavior with optional directives and clauses
    - **Optimize GPU occupancy**, register and shared memory usage, loop scheduling...
  - Loop schedule: spreading loop iterations over PEs of GPU
    - Compiler takes care of cases where iterations doesn't divide threadblock size
  - !$acc loop [gang] [worker] [vector]
    - Targets specific loop (or loops with collapse) at specific level of hardware
    - You can specify more than one
      - !$acc loop gang worker vector schedules loop iteration over all hardware

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)

parallel vs. kernels

- **parallel and kernels regions look very similar**
  - both define a region to be accelerated
    - different heritage; different levels of obligation for the compiler
  - parallel
    - prescriptive (like OpenMP programming model)
    - uses a single accelerator kernel to accelerate region
    - compiler will accelerate region (even if this leads to incorrect results)
  - kernels
    - descriptive
    - uses one or more accelerator kernels to accelerate region
    - compiler may accelerate region (if decides loop iterations are independent)
  - For more info: http://www.pgroup.com/lit/articles/insider/v4n2a1.htm

- **Which to use (our opinion)**
  - parallel (or parallel loop) offers greater control
    - **fits better with the OpenMP model**
  - kernels (or kernels loop) better for initially exploring parallelism
    - **not knowing if loopnest is accelerated could be a problem**

parallel loop vs. parallel and loop

- **parallel region can span multiple code blocks**
  - i.e. sections of serial code statements and/or loopnests
  - loopnests in **parallel** region are not automatically partitioned
    - **need to explicitly** use **loop** directive for this to happen
  - scalar code (serial code, loopnests without **loop** directive)
    - executed redundantly, i.e. identically by every thread
      - or maybe just by one thread per block (its implementation dependent)
  - There is no synchronization between redundant code or kernels
    - offers **potential for overlap of execution on GPU**
    - also offers **potential** (and likelihood) of **race conditions** and incorrect code
  - There is no mechanism for a barrier inside a parallel region
    - after all, CUDA offers no barrier on GPU across threadblocks
    - to effect a barrier, end the parallel region and start a new one
      - also use wait directive outside parallel region for extra safety

parallel loop vs. parallel and loop

- **My advice: don't...**
  - GPU threads are very lightweight (unlike OpenMP)
    - so don't worry about having extra **parallel** regions
  - explicit use of **async** clause may achieve same results
    - as using one **parallel** region
    - but with greater code clarity and better control over overlap

- **... but if you feel you must**
  - begin with composite **parallel loop** and get correct code
    - separate directives with care only as a later performance tuning
      - when you are sure the kernels are independent and no race conditions
    - this is similar to using OpenMP on the CPU
      - if you have multiple do/for directives inside omp parallel region
      - only introduce nowait clause when you are sure the code is working
      - and watch out for race conditions

Parallel Gotchas

- **No loop directive**
  - The code will (or may) run redundantly
    - Every thread does every loop iteration
    - Not usually what we want

```
!$acc parallel
  DO i = 1,N
   a(i) = b(i) + c(i)
  ENDDO
!$acc end parallel
```

- **Serial code in parallel region**
  - avoids copyin(t), but a good idea?
  - No! Every thread sets t=0
  - asynchronicity: no guarantee this finishes before loop kernel starts
  - race condition, unstable answers

```
!$acc parallel
  t = 0
!$acc loop reduction(+:t)
  DO i = 1,N
   t = t + a(i)
  ENDDO
!$acc end parallel
```

- **Multiple kernels**
  - Again, potential race condition
  - Treat OpenACC "end loop" like OpenMP "enddo nowait"

```
!$acc parallel
!$acc loop
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$acc loop
  DO i = 1,N
   a(i) = a(i) + 1
  ENDDO
!$acc end parallel
```

Sources of further information

- **OpenACC standard web page:**
  - OpenACC.org
    - documents: full standard and quick reference guide PDFs
    - links to other documents, tutorials etc.

- **Discussion lists:**
  - Cray users: openacc-users@cray.com
    - automatic subscription if you have a raven account
  - OpenACC forum: openacc.org/forum

- **CCE man pages (with PrgEnv-cray loaded):**
  - programming model and Cray extensions: intro_openacc
  - examples of use: openacc.examples
  - also compiler-specific man pages: crayftn, craycc, crayCC

- **CrayPAT man pages (with perftools loaded):**
  - intro_craypat, pat_build, pat_report
    - also command: pat_help
  - accpc (for accelerator performance counters)

OpenMP 4.0 main features

- **Target constructs for accelerator support**
  - OpenACC like functionality
  - Goal was to match OpenACC functionality, though there are some differences
- **SIMD**
  - Vectorization capability
- **Affinity**
  - Control of thread mapping
  - Portable support for –cc options
- **Cancellation**
  - Early exit from an OpenMP construct (search loop)
- **Task groups and dependencies**
  - Better control of task ordering and grouping
- **User Defined Reductions**
  - User created reduction folds (for example min/max with index)

omp target

- **omp target causes the region to run on the accelerator with a logical single thread**
  - the OpenACC equivalent to this is a top-level "acc parallel" region with "num_gangs(1)“

- **omp teams can only appear in "omp target" (with no statements in between)**
  - this is a fork-join parallelism construct, launching a "league of teams", but the teams are "loosely coupled"
    - the teams are not allowed to synchronize or make any assumptions about ordering of teams or progress of teams relative to one another
  - this is equivalent to the "gang" level in OpenACC

- **omp distribute is a loop worksharing construct that causes the "teams" (from an "omp teams" construct) to each execute a partition of the loop iterations**
  - since "teams" are loosely coupled, there is no implied barrier across teams at the end of this loop
  - This is equivalent to "acc loop gang”

omp target data

- A block-structured construct that defines a scope for creating device copies of host variables; the encountering thread remains executing on the host; **this is equivalent to "acc data"**

- map: a clause used on "target data" and "target" regions to specify which variables should be transferred to the device; this clause supports several "map types":
  - **alloc** (equivalent to OpenACC "pcreate");
  - **to** (equivalent to OpenACC "pcopyin");
  - **from** (equivalent to OpenACC "pcopyout");
  - **tofrom** (equivalent to OpenACC "pcopy").
  - **OpenMP only defines the "present_or" semantics**
    - OpenACC 2.5 is adopting this same behavior
  - OpenMP does not provide an equivalent OpenACC "present" clause
    - OpenMP 4.5 does

OpenACC to OpenMP - Compute Constructs

OpenACC	OpenMP
!$acc parallel	!$omp target teams
!$acc loop gang	!$omp distribute
!$acc loop worker	!$omp parallel do/for
!$acc loop vector	!$omp simd
!$acc loop gang vector	!$omp distribute simd
!$acc kernels	Not supported

OpenACC to OpenMP - Data regions

OpenACC

- !$acc data
  - create/pcreate
  - copyin/pcopyin
  - copy/pcopy
  - copyout/pcopyout
  - present(<list>)
- !$acc update self
- !$acc update device
- !$acc enter/exit data
- !$acc host_data

OpenMP

- !$omp target data
  - map(alloc:)
  - map(to:)
  - map(tofrom:)
  - map(from:)
  - map(<list>) (4.5)
- !$omp target update from
- !$omp target update to
- !$omp enter/exit target data (4.5)
- !$omp target data (4.5)
- map(always:[to:|from:|tofrom:] <list>) (4.5)

OpenACC to OpenMP - Separate Compilation

OpenACC	OpenMP
!$acc declare create	!$omp declare target
!$acc declare device_resident	Not supported
!$acc declare link	!$omp declare target link (4.5)
!$acc routine	!$omp declare target
!$acc routine(<name>)	!$omp declare target(<name>)

OpenACC to OpenMP - Other

OpenACC	OpenMP
● API routines	● Most supported in 4.5
● Atomics	● Use regular OpenMP atomics
● !$acc cache	● Not supported
● Async/Wait	● Tasks in 4.0 ● Depend/nowait on target in 4.5

A First OpenMP Device Program

```
PROGRAM main
  INTEGER :: a(N)
  <stuff>

!$omp target
!$omp teams
!$omp distribute
  DO i = 1,N
   a(i) = i
  ENDDO
!$omp end distribute
!$omp end teams
!$omp end target
!$omp target
!$omp teams
!$omp distribute
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$omp end distribute
!$omp end teams
!$omp end target

  <stuff>
END PROGRAM main
```

- **First loop nest initializes array**
- **Second loop nest modifies array**
- **Each loop nest is target region**
- **Compiler turns region into kernel**
- **teams creates threads**
  - divided into a "league of teams"
  - like CUDA "grid of threadblocks"
- **distribute partitions loop**
  - iterations divided over threads
- **Breaking target region gives barrier**
  - Only way to get global sync

A First Program

```
PROGRAM main
  INTEGER :: a(N)
  <stuff>


!$omp target teams distribute
  DO i = 1,N
   a(i) = i
  ENDDO
!$omp end target teams distribute

!$omp target teams distribute
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$omp end target teams distribute


  <stuff>
END PROGRAM main
```

- Data movements
  - Automatically scoped
- First kernel initializes array a(:)
  - Compiler chooses map(from:a)
- Second kernel modifies array a(:)
  - Compiler chooses map(tofrom:a)

- Note:
  - composite directives are shorter
    - target teams distribute
  - data sloshing of a(:)
    - moved to/from GPU between kernels

A Second Version

```
PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$omp target data map(from:a)

!$omp target teams distribute
  DO i = 1,N
   a(i) = i
  ENDDO
!$omp end target teams distribute

!$omp target teams distribute
DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$omp end target teams distribute

!$omp end target data
  <stuff>
END PROGRAM main
```

- Now add a **target data** region
  - to reduce data sloshing
- Specified arrays only move at boundaries of data region
- Unspecified arrays still moved by each kernel
- No automatic scoping for data regions

- **Data regions are the single biggest optimization in an offload code**
  - **Should be introduced at highest level in code possible**

A Second Version

```
PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$omp target data map(from:a)

!$omp target teams distribute
  DO i = 1,N
   a(i) = i
  ENDDO
!$omp end target teams distribute

!$omp target teams distribute
DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$omp end target teams distribute

!$omp end target data
  <stuff>
END PROGRAM main
```

- **target data** regions
  - contain **target** region(s) and (optionally) host code
  - can be nested
- Two copies of arrays inside region
  - One on host
  - One on accelerator
- Copies of arrays independent
  - No automatic synchronization of copies within data region
    - Only at the boundaries
- **target update** directive
  - user-directed synchronization within **target data** region

Data Movement Clauses

- **Applied to: target data, target**

- **map(<maptype>:<array>)**
  - **to**, **from**, **tofrom**
    - Moves data "to" GPU at start of region and/or "from" GPU at end
    - Supply list of arrays or array sections (using ":" notation)
    - Fortran uses start:end; C/C++ uses start:length
      - e.g. first N elements: Fortran 1:N (familiar); C/C++ 0:N (less familiar)
      - Advice: be careful and don't make mistakes! ☺
      - **Use profiler and/or runtime commentary to see how much data moved**
      - Avoid **non-contiguous array slices** for performance
  - **alloc**
    - No copying, useful for shared temporary arrays in loop nests

  - **private, firstprivate:** as in host OpenMP
    - scalars (including loop variables) shared by default
      - **Advice: declare them anyway, for clarity**

Sharing GPU Data Between Subprograms

```
PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$omp target data map(from:a)
!$omp target teams distribute
  DO i = 1,N
   a(i) = i
  ENDDO
!$omp end target teams distribute
  CALL double_array(a)
!$omp end target data
  <stuff>
END PROGRAM main
```

```
SUBROUTINE double_array(b)
  INTEGER :: b(N)
!$omp target teams distribute
  DO i = 1,N
   b(i) = double_scalar(b(i))
  ENDDO
!$omp end target teams distribute
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)
  - Compiler supports function calls inside **target** regions
- Array b(:) will be scoped as **map(tofrom:b)** [automatically or explicitly]
  - Compiler will always first check if the array is already on GPU
  - If so, will use that version and not copy the data
- **Original calltree structure of program can be preserved**

CUDA Interoperability (OpenMP4.5)

```fortran
PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$omp target data map(from:a)
! <Populate a(:) on device
!  as before>
!$omp target data use_device_ptr(a)
  CALL dbl_cuda(a)
!$omp end target data
!$omp end target data
  <stuff>
END PROGRAM main
```

```c
__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();
}
```

- **use_device_ptr region exposes accelerator memory address**
  - on inner target data region (nested inside outer target data region)
- **CUDA-C wrapper compiled with nvcc linked with CCE)**
  - Must include cudaThreadSynchronize() before and after
    - Before: so asynchronous accelerator kernels definitely finished
    - After: so CUDA kernel definitely finished
  - CUDA kernel written as usual
  - Can use same method to call existing CUDA library or G2G-enabled MPI

Clauses for !$omp target teams distribute

- **Tuning clauses:**
  - User can tune default behavior with optional directives and clauses
  - Loop schedule: spreading loop iterations over accelerator threads
    - Compiler takes care when # iterations doesn't divide threadblock size
  - num_teams, thread_limit:
    - Control the number of threadblocks, threads per block (CCE default: 128)
    - Only need num_teams for tuning
  - collapse: Apply tuning to multiple loops
  - dist_schedule: Control mapping of loop iterations to threads
  - nowait, depend: Create asynchronous tree (DAG) of tasks

- **Other clauses:**
  - if: runtime decision to execute loopnest on host or device
  - reduction: specify reduction variables (as in traditional OpenMP)

More OpenMP device directives

- **Performance tuning**
  - **!$omp simd**
    - Choose which loop (or loops with **collapse**) to vectorize in loop nest

- **Data synchronisation**
  - **!$omp target update** [**to**|**from**]
    - Copy specified arrays (slices) within **target data** region
  - Useful if you only need to send a small subset of data **to/from** accelerator
    - e.g. halo exchange for domain-decomposed parallel code
    - or sending a few array elements to the CPU for printing/debugging
  - **Remember slicing syntax differs between Fortran and C/C++**
  - **The array sections should be contiguous**
  - Can also use **nowait**, **depend** tuning clauses for asynchronous DAG

- **!$omp declare target**
  - Makes a variable resident in accelerator memory
    - persists for the duration of the implicit data region
  - Also used to execute subprogram on the accelerator (avoiding inlining)

Directives: composite or separate?

- **!$omp target teams distribute or separate directives**
- **Separate directives allow larger target regions**
  - Spanning several loop nests
  - But with a huge potential for race conditions

- **Composite is always the best starting point**
  - GPU threads are very lightweight (unlike host OpenMP)
    - so don't worry about having extra **teams** regions
  - explicit use of **nowait** clause may achieve same results
    - as using one large **target** region
    - but with greater code clarity and better control over overlap

- **... but if you feel you must**
  - begin with composite version and get correct code
    - separate directives with care only as a later performance tuning
      - when you are sure the kernels are independent and no race conditions

Sources of further information

- **OpenMP standard web page:**
  - OpenMP.org
    - documents: full standard and quick reference guide PDFs
    - links to other documents, tutorials etc.

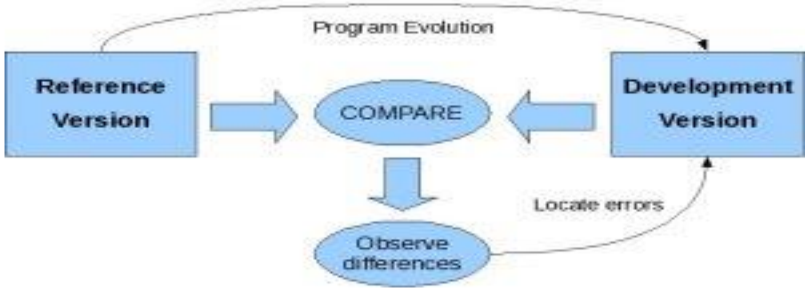
- **Discussion lists:**
  - OpenMP forum: openmp.org/forum

- **CCE man pages (with PrgEnv-cray loaded):**
  - compiler-specific man pages: crayftn, craycc, crayCC

- **CrayPAT man pages (with module perftools-base loaded):**
  - intro_craypat, pat_build, pat_report
    - also command: pat_help
  - accpc (for accelerator performance counters)

CCDB Overview

- **What is comparative debugging?**
  - Data centric approach instead of the traditional control-centric paradigm
  - Two applications, same data
  - Key idea: The data should match
  - Quickly isolate deviating variables
- **Comparative debugging tool**
  - NOT a traditional debugger!
  - Assists with comparative debugging
  - CCDB GUI hides the complexity and helps automate process
    - Creates automatic comparisons
    - Based on symbol name and type
    - Allows user to create own comparisons
    - Error and warning epsilon tolerance
    - Scalable
- **How does this help me?**
  - Algorithm re-writes
  - Language ports
  - Different libraries/compilers
  - New architectures
- **Collaboration with University of Queensland**

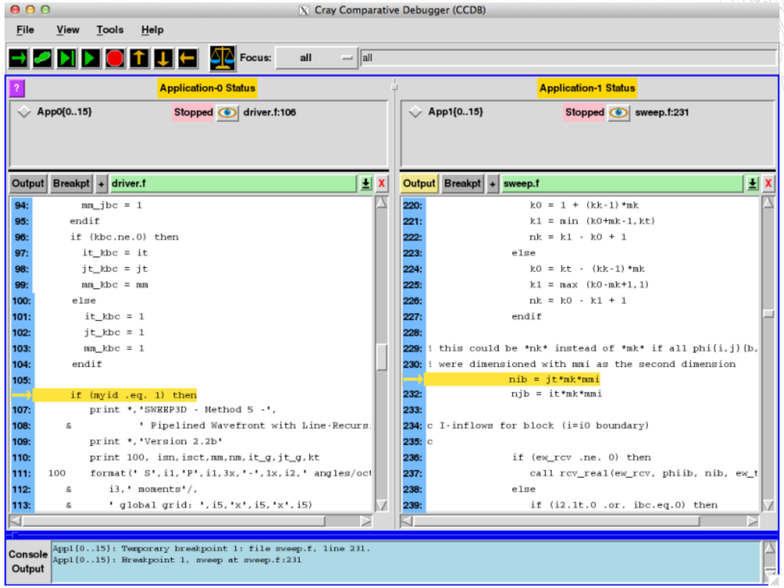


THE UNIVERSITY OF QUEENSLAND AUSTRALIA

SC'15 – L. DeRose et al. "Relative Debugging for a Highly Parallel Hybrid Computer System"

Comparative Debugging

- **Specify conditions for correct behavior prior to execution**
- **Debugger:**
  - keeps track of comparison points (breakpoints)
  - performs comparison automatically
- **Control returned to user:**
  - examination of state
  - continuation of execution



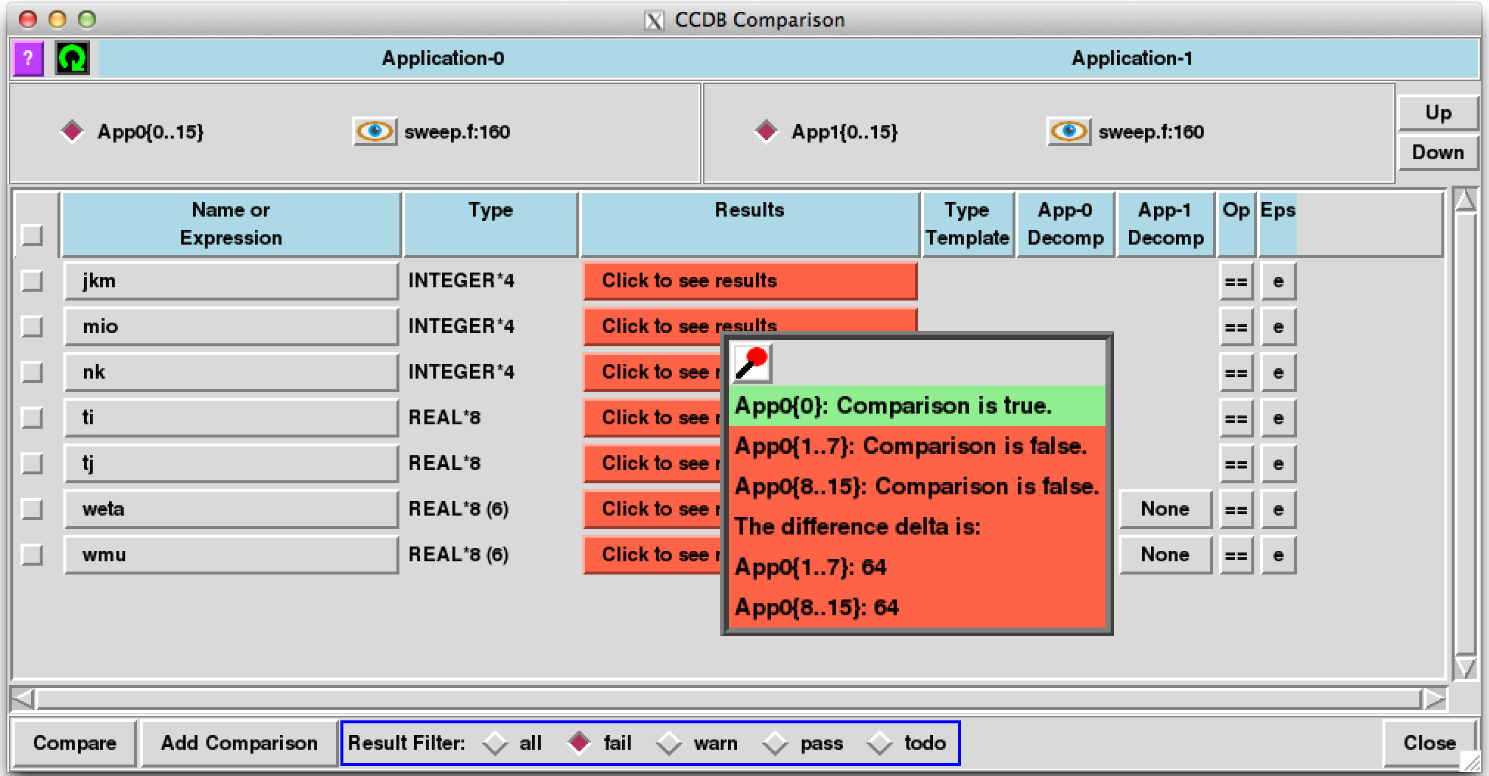
```
          mm_jbc = 1
        endif
        if (kbc.ne.0) then
          it_kbc = it
          jt_kbc = jt
          mm_kbc = mm
        else
          it_kbc = 1
          jt_kbc = 1
          mm_kbc = 1
        endif

        if (myid .eq. 1) then
            print *,'SWEEP3D - Method 5 -',
     &              ' Pipelined Wavefront with Line-Recurs
            print *,'Version 2.2b'
            print 100, isn,isct,mm,nm,it_g,jt_g,kt
 100        format(' S',i1,'P',i1,3x,'-',1x,i2,' angles/oct
     &         i3,' moments'/,
     &         ' global grid: ',i5,'x',i5,'x',i5)
```

```
                 k0 = 1 + (kk-1)*mk
                 k1 = min (k0+mk-1,kt)
                 nk = k1 - k0 + 1
              else
                 k0 = kt - (kk-1)*mk
                 k1 = max (k0-mk+1,1)
                 nk = k0 - k1 + 1
              endif

! this could be *nk* instead of *mk* if all phi(i,j){b,
! were dimensioned with mmi as the second dimension
              nib = jt*mk*mmi
              njb = it*mk*mmi

c I-inflows for block (i=i0 boundary)
c
              if (ew_rcv .ne. 0) then
                 call rcv_real(ew_rcv, phiib, nib, ew_t
              else
                 if (i2.lt.0 .or. ibc.eq.0) then
```

```
App1{0..15}: Temporary breakpoint 1: file sweep.f, line 231.
App1{0..15}: Breakpoint 1, sweep at sweep.f:231
```

assert P1::T1[0..99]@”file.c”:240 = P2::Y2(1,100)@”prog.f”:300

CCDB - Comparison



CCDB Comparison

	Application-0		Application-1	
	App0{0..15}	sweep.f:160	App1{0..15}	sweep.f:160

	Name or Expression	Type	Results	Type Template	App-0 Decomp	App-1 Decomp	Op	Eps
	jkm	INTEGER*4	Click to see results				==	e
	mio	INTEGER*4	Click to see results				==	e
	nk	INTEGER*4	Click to see r				==	e
	ti	REAL*8	Click to see r				==	e
	tj	REAL*8	Click to see r				==	e
	weta	REAL*8 (6)	Click to see r			None	==	e
	wmu	REAL*8 (6)	Click to see r			None	==	e

App0{0}: Comparison is true.
App0{1..7}: Comparison is false.
App0{8..15}: Comparison is false.
The difference delta is:
App0{1..7}: 64
App0{8..15}: 64

Result Filter: all [✓ fail] warn pass todo

Introduction to the Cray Accelerated Scientific Libraries

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

Cray Adaptive Scientific Libraries

- **The Cray Scientific Libraries have three concentrations to increase productivity with enhanced performance**
  - **Standardization**
  - **Autotuning**
  - **Adaptive Libraries**

- **Cray adaptive model based on autotuning**
  - Runtime analysis allows **best** library/kernel to be **used dynamically**
  - Extensive offline testing allows **library to make decisions** or remove the need for those decisions
  - Decision depends on the system, on previous performance info, obtained previously, and characteristics of calling problem

What Makes Cray Libraries Special

- **Cray scientific libraries are designed to give maximum possible performance from Cray systems with minimum effort**
  - **Node performance**
    - Highly tuned BLAS etc at the low-level
  - **Network performance**
    - Optimize for network performance
    - Overlap between communication and computation
    - Use the best available low-level mechanism
    - Use adaptive parallel algorithms
  - **Highly adaptive software**
    - Using auto-tuning and adaptation, give the user the known best (or very good) codes at runtime
  - **Productivity features**
    - Simpler interfaces into complex software

What is Cray Libsci_acc?

- **Provide basic scientific libraries optimized for hybrid systems**
  - Incorporate the existing GPU libraries into Cray libsci

- **Independent to, but fully compatible with OpenACC and OpenMP 4.0**

- **Multiple use case support**
  - Get the base use of accelerators with no code change
  - Get extreme performance of GPU with or without code change

- **Provide additional performance and usability**

- **Three interfaces**
  - Simple interface
    - **Auto-adaptation**
    - Base performance of GPU with minimal (or no) code change
    - Target for anybody: non-GPU users and non-GPU expert

  - Expert interfaces (Device and CPU)
    - Advanced performance of the GPU with controls for data movement
    - Target for CUDA, OpenACC, and GPU experts
      - **Does not imply the expert interfaces are always needed to get great performance**

Why libsci_acc ?

- **Several scientific library packages are already there**
  - CUBLAS,
  - CUFFT,
  - CUSPARSE (NVIDIA),
  - MAGMA (U Tennessee),
  - CULA (EM Photonics).

- **Code modification is required to use these existing GPU libraries!**
  - No Compatibility to Legacy APIs
    - cublasDgemm(….)
    - magma_dgetrf(…)
    - culaDgetrf(…)
    - Why not dgemm(), dgetrf()?

  - Not focused on Standard API (Fortran, C, C++)
    - Require CUDA data types, primitives and functions in order to call them

- **Performance**

Usage – Basics

- **Fortran and C interfaces (column-major assumed)**
  - Load the module craype-accel-nvidia35
  - Compile as normal (dynamic libraries used)
- **To enable threading in the CPU library, set OMP_NUM_THREADS**
  - e.g. export OMP_NUM_THREADS=16
- **Assign 1 single MPI process per node**
  - Multiple processes cannot share the single GPU
- **Execute your code as normal**

Three interfaces for three use cases

- Simple interface

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

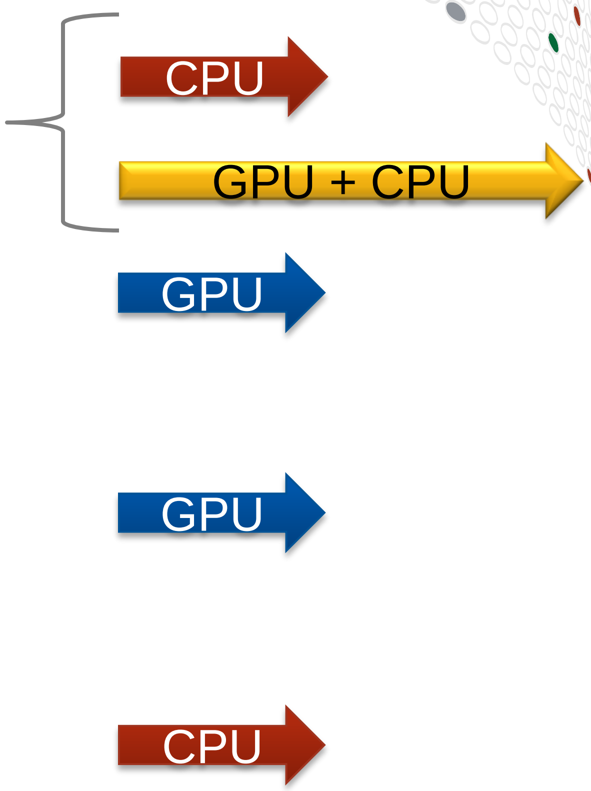
```
dgetrf(M, N, d_A, lda, ipiv, &info)
```

- Device interface

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

- CPU interface

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



Adaptation in the Simple Interface

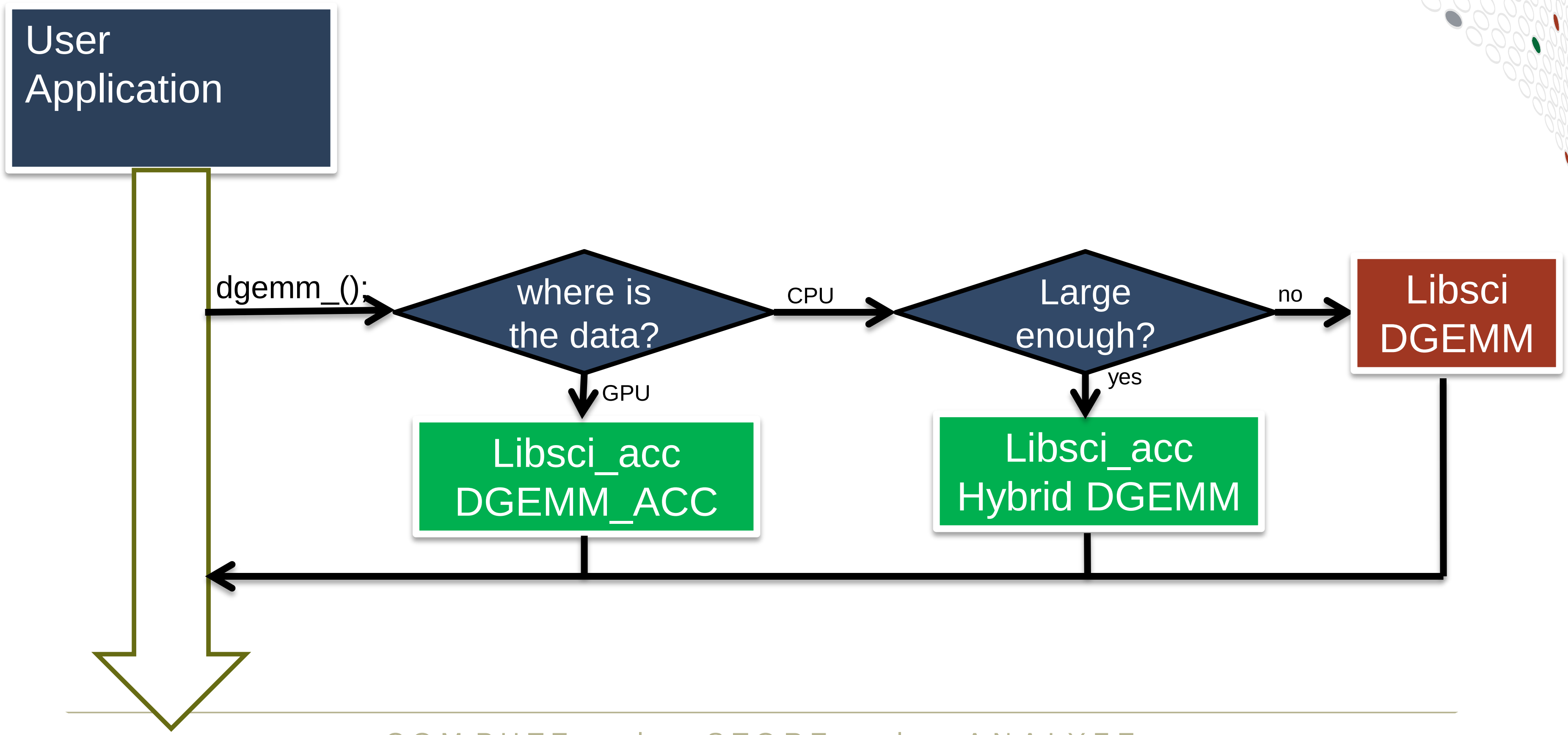
- You can pass either host pointers or device pointers with the simple interface

- A is in host memory
  - dgetrf(M, N, A, lda, ipiv, &info)
  - Performs hybrid operation on GPU
  - if problem is too small, performs host operation

- Pass Device memory
  - dgetrf(M, N, d_A, lda, ipiv, &info)
  - Performs hybrid operation on GPU

- **BLAS 1 and 2 performs computation local to the data location**
  - **CPU-GPU data transfer is too expensive to exploit hybrid execution**

Libsci_acc: Simple Interface for BLAS3 and LAPACK



Device interface

- **Device interface gives higher degrees of control**
- **Requires that you have already copied your data to the device memory**
- **API**
  - Every routine in libsci has a version with _acc suffix
  - E.g. dgetrf_acc
  - This resembles standard API except for the suffix and the device pointers

CPU interface

- **Sometimes apps may want to force ops on the CPU**
  - Need to preserve GPU memory
  - Want to perform something in parallel
  - Don’t want to incur transfer cost for a small op
- **Can force any operation to occur on CPU with _cpu version**
- **Every routine has a _cpu entry-point**
- **API is exactly standard otherwise**

libsci_acc interaction with OpenMP/OpenACC

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

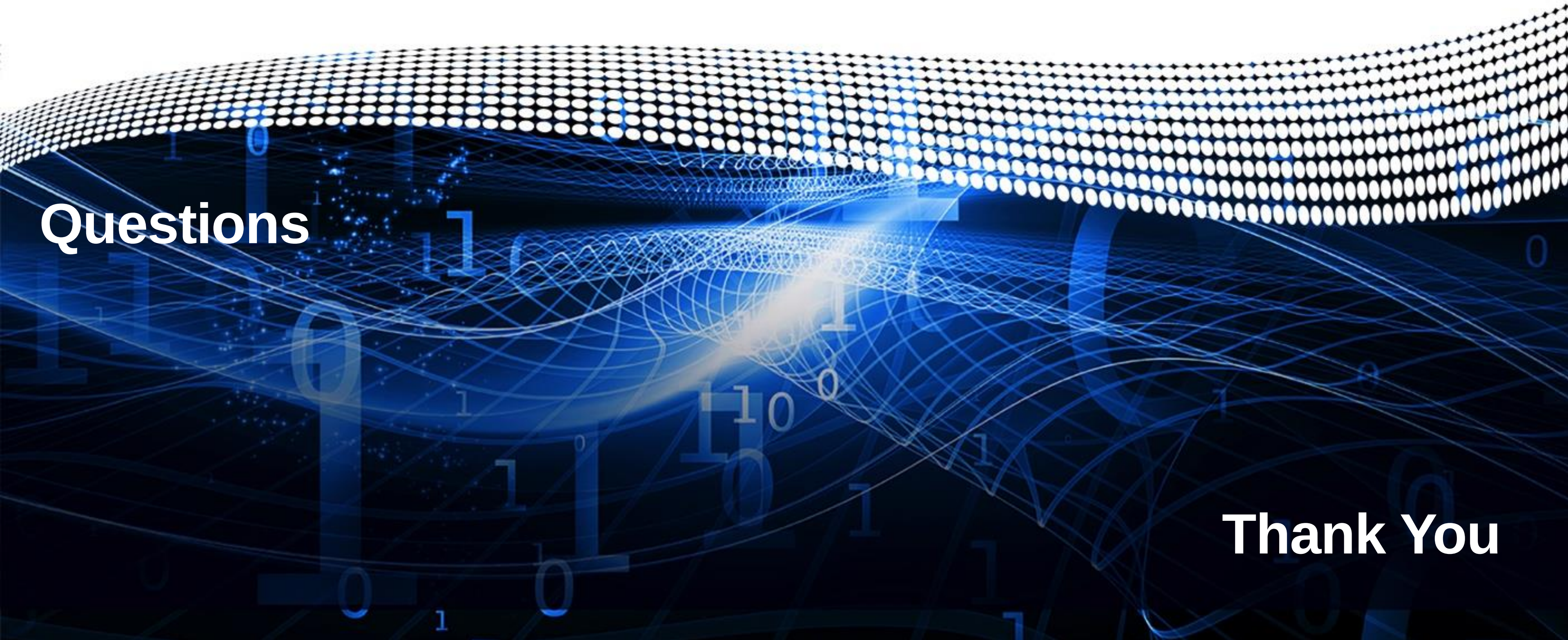
```
!$acc data copy(a,b,c)

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

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

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

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

libsci_acc interaction with OpenMP/OpenACC

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

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

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

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

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

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

COMPUTE | STORE | ANALYZE



Questions

Thank You