OpenACC: Productive, Portable Performance on Hybrid Systems Using High-Level Compilers and Tools

Luiz DeRose, Alistair Hart, Heidi Poxon, & James Beyer

Cray Inc.

Goals of this Tutorial

- Motivate why directive-based programming of accelerators is useful
- Introduce the OpenACC high level parallel programming model for porting and developing applications to hybrid systems with accelerators attached to multi-core processors;
- Present debug and optimization strategies through use of tools
- Provide hands-on experience in using OpenACC directives, development guidance, practical tricks, and tips, so these systems can be used effectively
- The idea is to equip you with the knowledge to develop applications that run efficiently on parallel hybrid supercomputers
  - Not just on single GPU

Structure of this Tutorial

- **Aims to lead you through the entire development process**
  - What is OpenACC?
  - How do I use it in a simple code?
  - How do I port a real-sized application?
  - Performance tuning and advanced topics
  - Case studies
- **It will assume you know**
  - Basic understanding of parallel computing and programming models
  - Knowledge of a scientific programming language such as Fortran, C, or C++
  - Basic understanding of Linux and use of an editor
- **It will help if you know**
  - A little bit about GPU architecture and programming
    - SMs, threadblocks and warps, coalescing
  - The basic idea behind OpenMP programming
    - but this is not essential

Outline

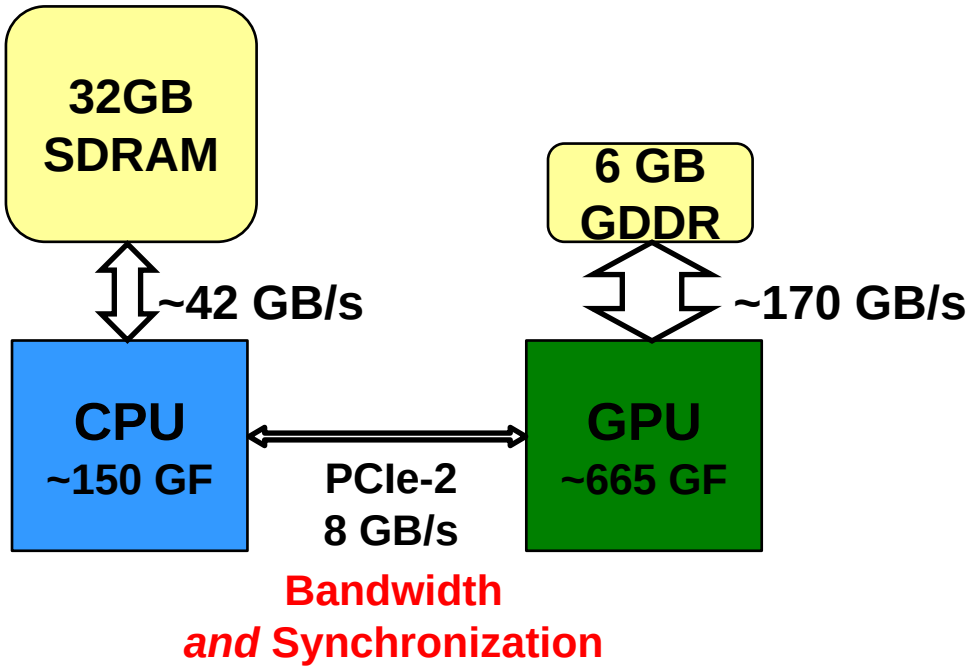
08:30-09:00	Introduction & Motivation
09:00-09:45	The OpenACC programming model
09:45-10:00	***Practical 0: Getting started with a simple test code***
10:00-10:30	Break
10:30-11:00	Porting tools
11:00-11:30	Porting a simple code
11:30-12:00	***Practical 1: Porting a simple code yourself***
12:00-13:30	Lunch
13:30-13:50	OpenACC performance tuning
13:50-14:10	Performance tools for OpenACC
14:10-14:30	Porting a larger code
14:30-15:00	***Practical 2: Porting the larger code yourself***
15:00-15:30	Break
15:30-15:55	Advanced topics
15:55-16:10	Debugging OpenACC codes
16:10-16:35	Porting a parallel code to OpenACC
16:35-17:00	Roadmap, outlook, and conclusions

The New Generation of Supercomputers

- **Hybrid multicore has arrived and is here to stay**
  - Wide nodes are getting wider
  - Accelerators have leapt into the Top500

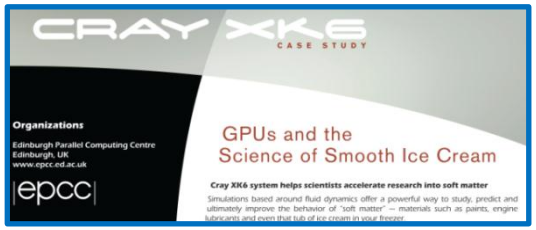
- **Programming accelerators efficiently is hard**
  - Three levels of parallelism required
    - MPI between nodes or sockets
    - Shared memory programming on the node
    - Vectorization for low level looping structures
  - Need a hybrid programming model to support these new systems
  - Need a high level programming environment
    - Compilers, tools, & libraries

Structural Issues with Accelerated Computing

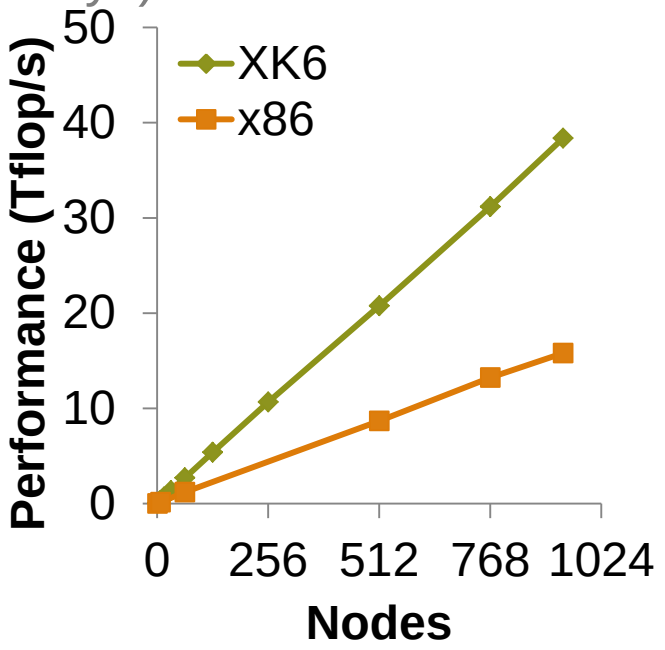


- **Trick is to keep kernel data structures resident in GPU memory as much as possible**
  - **Avoid copying** between CPU and GPU
  - **Use asynchronous, non-blocking, communication, multi-level overlapping**

CUDA on Cray XK6



- **If you work hard, you can get good parallel performance**
- **Ludwig Lattice Boltzmann code rewritten in CUDA**
  - Reordered all the data structures (structs of arrays)
  - Pack halos on the GPU
  - Streams to overlap compute, PCIe comms, MPI halo swaps
- **10 cabinets of Cray XK6**
  - 936 GPUs (nodes)
- **Only 4% deviation from perfect weak scaling between 8 and 936 GPUs**



- **Most scientific applications will not have this level of developer support (Ludwig was special research case)**

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**
- **Accelerated programming needs 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

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 OpenACC Version 1.0 complete in 2012**
  - Cray CCE: Version 2.0 complete in September 2013 (CCE 8.2)
  - PGI Accelerator version 12.6 onwards
  - CAPS Full support in version 1.3

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)

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
```

Reduction Code Summary

Summary of code complexity and performance

Programming Language / 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

Strategic Risk Factors

- **Will there be machines to run my OpenACC code on?**
  - Now? Lots of Nvidia GPU accelerated systems
    - Cray XC30: CSCS Piz Daint, …
    - Cray XK7s: ORNL Titan, NCSA Blue Waters, CSCS Tödi, HLRS Hermit, ...
    - Lots of other GPU machines in Top100 (OpenACC is multi-vendor)
  - Future? OpenACC can be targeted at other accelerators
    - PGI already target Intel Xeon Phi, AMD GPUs
  - Plus you can always run on CPUs using same codebase

- **Will OpenACC continue?**
  - Support? Cray and PGI (at least) are committed to support OpenACC
    - Lots of big customer pressure to continue to run OpenACC
  - Develop? OpenACC committee finalized v2.0 of standard and working on next version
    - Lots of new partners joined committee at end of last year (last count was 15)

- **Will OpenACC be superseded by something else?**
  - Auto-accelerating compilers? If only!
    - Never really managed it for threading real HPC applications on the CPU
    - Data locality adds to the challenge
  - OpenMP accelerator directives? OpenACC work not wasted
    - Very similar programming model; can transition if wish

The OpenACC Programming Model

Contents

- What is OpenACC?
- How does it work?
  - The execution and memory models
- What does it looks like?
- How do I use it?
  - Basic directives
    - Enough to do the first two practicals
    - Advanced topics will follow in another lecture
- Where can I learn more?

- Plus a few hints, tips, tricks and gotchas along the way
  - Not all guaranteed to be relevant, useful (or even true)

OpenACC Execution model, or who's in charge

- **In short:**
  - If you've used CUDA, it's just like that but more automated
- **In more detail:**
  - The "host" (i.e. the CPU) is in charge
    - The main program executes on the host
    - The host can offload tasks to the "device" (accelerator, e.g. attached GPU)
      - tasks can be computation or data transfer between host and device
    - The host is responsible for managing the accelerator memory
      - as well as its own; allocating and freeing memory as needed
    - The host is responsible for synchronisation
      - ensuring offloaded tasks have completed
  - The difference with OpenACC is that a lot of this is handled automatically by the compiler and/or runtime system

OpenACC Execution model

- **In much more detail:**
  - Host-directed execution with attached accelerator (e.g. 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

- **In short:**
  - If you've used CUDA, it's just like that but more automated
- **In more detail:**
  - The host and device have separate memories
    - There's no automatic data synchronisation going on in the background
  - GPU memory is fragmented and there is no mechanism for sharing data between all threads during computation
    - which leaves potential for race conditions
  - The difference with OpenACC is that the compiler and runtime help:
    - handle some of the memory synchronisation tasks
    - can avoid or identify some race conditions
      - but never all of them

OpenACC Memory model

- **In much more detail:**
  - Memory spaces on the host and device distinct
    - Different locations, possibly different address space
    - Data movement performed by host using runtime library calls that explicitly move data between the separate
  - GPUs have a weak memory model
    - No synchronisation between different execution units (SMs)
      - Unless explicit memory barrier
    - 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.

Accelerator directives

- **Modify original source code with directives**
  - Non-executable statements (comments, pragmas)
    - Can be ignored by non-accelerating compiler
    - CCE -hnoacc (or -xacc) also supresses compilation
  - Sentinel: acc
    - C/C++: preceded by #pragma
      - Structured block {...} avoids need for end directives
    - Fortran: preceded by !$ (or c$ for FORTRAN77)
      - Usually paired with !$acc end *
      - Directives can be capitalized
  - Continuation to extra lines allowed
    - C/C++: \ (at end of line to be continued)
    - Fortran:
      - Fixed form: c$acc& or !$acc& on continuation line
      - Free form: & at end of line to be continued
        - continuation lines can start with either !$acc or !$acc&

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

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

Conditional compilation

- **In theory, OpenACC code should be identical to CPU**
  - only difference are the directives (i.e. comments)
- **In practise, you may need slightly different code**
  - For example, to cope with:
    - calls to OpenACC runtime API functions
    - where you need to recode for OpenACC
      - such as for performance reasons
      - you should try to minimise this
        - usually better OpenACC code is better CPU code
- **CPP macro defined to allow conditional compilation**
  - _OPENACC == yyyymm
    - Version 1.0: 201111
    - Version 2.0: 201306

A first example

Execute a loop nest on the GPU

- **Compiler does the work:**
  - Data movement
    - allocates/frees GPU memory at start/end of region
    - moves of data to/from GPU

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

write-only

read-only

  - Loop schedule: spreading loop iterations over PEs of GPU
    - **OpenACC** **CUDA**
    - `gang`: a threadblock
    - `worker`: warp (group of 32 threads)
    - `vector`: threads within a warp
    - Compiler takes care of cases where iterations doesn't divide threadblock size
  - Caching (e.g. explicit use GPU shared memory for reused data)
    - automatic caching can be important
  - Tune default behavior with optional clauses on directives

OpenACC suitability

- **Will my code accelerate well with OpenACC?**
  - Computation should be based around loopnests processing arrays
    - Loopnests should have defined tripcounts (either at compile- or run-time)
      - while loops will not be easy to port with OpenACC
      - because they are hard to execute on a GPU
    - Data structures should be simple arrays
      - derived types, pointer arrays, linked lists etc. may stretch compiler capabilities
  - The loopnests should have a large total number of iterations
    - at least measured in the thousands
      - even more is better; less will execute, but with very poor efficiency
  - The loops should span as much code as possible
    - maybe with some loops very high up the callchain
  - The loopnest kernels should not be too branched
    - one or two nested IF-statements is fine
    - too many will lead to slow execution on many accelerators
  - The code can be task-based
    - but each task should contain a suitable loopnest

Accelerator kernels

- **We call a loopnest that will execute on the GPU a "kernel"**
  - this language is similar to CUDA
  - the loop iterations will be divided up and executed in parallel
- **We have choice of two directives to create a kernel**
  - parallel loop or kernels loop
    - both generate an accelerator kernel from a loopnest
      - the language is confusing
  - Why are there two and what's the difference?
    - You can use either
    - We'll discuss the difference at the end of this lecture
    - This tutorial concentrates on using the parallel loop directive
- **Actually, these are composite directives**
  - parallel loop combines the parallel and loop directives, as a short-cut
  - you can use the directives separately, but
    - you need to have a good reason and be careful
      - or you can get incorrect answers
    - we'll go into detail on this later in the tutorial

A first full 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 initialises array
    - Compiler will determine copyout(a)
  - Second kernel updates array
    - Compiler will determine copy(a)
  - Breaking parallel region=barrier
    - No barrier directive (global or within SM)

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

A second version

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

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

- **No automatic synchronisation of copies within data region**
  - User-directed synchronisation via update directive

Sharing GPU data between subprograms

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

```
SUBROUTINE double_array(b)
  INTEGER :: b(N)
!$acc parallel loop present(b)
  DO i = 1,N
   b(i) = double_scalar(b(i))
  ENDDO
!$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 for OpenACC 1.0
    - Compiler will inline (maybe need `-Oipafrom` or program library)
  - All OpenACC 2.0 compilers will support call inside of compute constructs
- 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
- Original calltree structure of program can be preserved

Data scoping

- **Data scoping**
  - We want the same answer when executing in parallel as when serially
- **We declare variables in a loopnest to be shared or private**
  - shared
    - all loop iterations all process the same version of the variable
    - variable could be a scalar or an array
    - a and b are shared arrays in this example
  - private
    - each loop iteration uses variable separately
    - again, variable could be a scalar or an array
    - t is a private scalar in this example
    - loop index variables (like i) are also private
  - firstprivate: a variation on private
    - each thread's copy set to initial value
    - loop limits (like N) should be firstprivate

```
for (i=0; i<N; i++) {
   t = a[i];
   t++;
   b[i] = 2*t;
}
```

Data scoping in OpenACC

- In OpenMP, we have exactly these data clauses
  - shared, private, firstprivate
  - loop index variables private by default (unless we say otherwise)
- In OpenACC
  - private, firstprivate are just the same
  - scalars *and* loop index variables are private by default
    - in parallel regions, but *NOT* in data regions
  - shared variables are more complicated in OpenACC
    - because we also need to think about data movements to/from GPU
  - copyin: a shared variable that is used read-only in the loopnest
  - copyout: a shared variable that is used write-only in the loopnest
  - copy: a shared variable that is used read-write in the loopnest
  - create: a shared variable that is a temporary in the loopnest
    - (although there is still an unused copy on the host in this case)

Reduction variables

- **Reduction variables are a special case of private variables**
  - where we will need to combine values across loop iterations
  - e.g. sum, max, min, logical-and etc.

- **We need to tell the compiler to treat this appropriately**
  - Use the reduction clause for this (added to parallel loop directive)
    - same expression in OpenACC as in OpenMP
  - Examples:
    - sum: use clause reduction(+:t)
      - Note sum could involve adding and/or subtracting
    - max: use clause reduction(max:u)

```
DO i = 1,N
   t = t + a(i) - b(i)
   u = MAX(u,a(i))
ENDDO
```

  - Some things to remember
    - OpenACC only allows reductions of scalar variables, not of array elements
      - try using a temporary scalar in the loopnest for the reduction instead
    - You need to put a reduction clause on every relevant loop directive
      - don't worry about this now
      - but it will be more important later when we start performance tuning

Data clauses in detail

- **Data clauses are applied to:**
  - accelerated loopnests: parallel and kernels directives
    - here they over-ride relevant parts of the automatic compiler analysis
    - you can switch off all automatic scoping with default(none) clause (in v2)
  - data regions: data directive (plus enter/exit data in OpenACC v2)
    - Note there is no automatic scoping in data regions (arrays or scalars)

- **Shared clauses (copy, copyin, copyout, create)**
  - supply list of scalars, arrays or array sections

- **Private clauses (private, firstprivate, reduction)**
  - only apply to accelerated loopnests (parallel and kernels directives)

Array sections and unshaped pointers

- **Data clauses can accept array section arguments**
  - array sections specified using ":" notation
  - sections must be contiguous
    - only specify incomplete section on the fastest-moving array index
      - Left-most for Fortran, right-most for C/C++
  - syntax differs slightly between languages
    - Fortran uses start:end, so first N elements is a(1:N)
    - C/C++ uses start:length, so first N elements is b[0:N]
    - Advice: be careful when switching languages!
      - Use profiler, runtime commentary to see how much data moved

- **Fortran arrays have a complicated data structure**
  - descriptor/dope vector that contains information about size and shape
- **Arrays in C/C++ are often just pointers**
  - How many bytes should be transferred here: copy(c) ?
  - In C/C++ you usually need to be more explicit: copy(c[0:N])
    - otherwise the compiler or runtime will complain (CCE: "unshaped pointer")

Sharing data between kernels

- **If you are using a data region around kernels**
  - Must ensure that the runtime uses the shared data already present
- **If the kernel is in the same routine as the data region**
  - Then just don't mention those bits of data in the parallel clauses
- **If the kernel is in a different routine to the data region**
  - On the parallel or kernels directive:
    - Specify the relevant data with present clause
    - instead of other shared clauses (copy, copyin, copyout, create)
    - don't rely on automatic scoping for shared data in this case
- **If an array is declared present, but is not on accelerator**
  - you get a runtime error and the program crashes

Controlling whether accelerator is used

- **What if I call a routine with kernels in two ways**
  - sometimes inside data region, when data should be present
  - sometimes outside a data region, when data should be copied
- **In both cases I want to process data on accelerator**

- **To do this, instead of using present clause**
  - on parallel or kernels directives, use:
    - present_or_copy*, present_or_create
    - or the short form: replace present_or_ by p, e.g. pcopyin, pcreate
  - The runtime will check if data is already on the device
    - if so, it uses that version (with no data copying)
    - if not, it does the prescribed data copying either side of the kernel

  - Advice: only use present_or_* versions if you really must
    - most "real" codes have a simple calltree:
      - data to be processed on the accelerator should always be present
      - "not present" runtime errors are a useful debugging tool

Controlling whether accelerator is used

- **if clause: applied to parallel or kernels directives**
  - decide at runtime whether to execute loopnest on CPU or accelerator
    - executes on GPU if TRUE, otherwise on CPU

- **Advice: this is less useful than you may think**
  - OpenACC performance is all about data locality
  - If data could either be on the CPU or the GPU, really want to:
    - process on accelerator if data is on accelerator
    - process on CPU if data is on the CPU

  - To do this, need to use OpenACC runtime API to check location, then
    - either have two versions of the routine, one with OpenACC, one without
      - and explicitly branch code, based on data location
    - or use the result of the API call as the argument to the if clause

  - We'll cover the runtime API calls later in the tutorial
    - But maybe you should first think: could data always be on accelerator?

Updating data

- **Data regions are used to keep data on accelerator**
  - can span multiple accelerator kernels
  - can also include serial code

- **the data clauses create *copies* of data arrays**
  - exist on accelerator for duration of data region
  - the host versions of the array continue to exist, and can be used
  - host, accel copies are only synchronised at start/end of data region

  - You can synchronise copies manually within a data region
    - for instance:
      - to copy a halo buffer back to the host for communication
      - to copy values of an array to the CPU for checking or printing
    - You do this using the update directive, for instance:
      - update host(a,b) copies entire arrays a,b from accelerator to host
      - update device(c(j:k)) copies a slice of array c from host to accelerator
        - note the different slice syntax for C/C++
        - watch out: c(3) is one element of the array, not the first 3 elements
      - slices should be contiguous in memory
        - so only use slice notation for the fastest-moving array index

And take a breath...

- **You now know everything you need to start accelerating**
  - This is all you need to do the first practical
  - Just using what you know, the scalar Himeno code:
    - is fully ported to the GPU
    - runs faster on the GPU than it does across 16 cores of the CPU
- **So what do we do for the rest of the tutorial?**
  - Not all codes are as simple as the scalar Himeno code
  - OpenACC has a lot more detail
    - performance tuning options
    - additional functionality
- **Before we finish, a quick word on parallel and kernels...**

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 (like PGI Accelerator programming model)
    - 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 (just an 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

Summary

- **We introduced the OpenACC programming model**
  - the main program executes on the host (i.e. the CPU)
  - host sends computational tasks to the device (accelerator, GPU)
    - computational kernels or data transfers
  - the host handles synchronisation (automatically or under user control)
- **We've described the sorts of codes that execute well**
- **We've seen the basic directives used to OpenACC codes**
  - parallel loop (or kernels loop) to create kernels
  - data regions to minimise unnecessary data movement
    - data clauses used to control data scope and movement
    - update directive for extra synchronisation in data regions

Porting Tools

Contents

- **This lecture is all about finding as much parallelism as you can in your application**
  - Why should you do this?
  - How can you do this?
  - What help can you get?
- **How can a hybrid code benefit you**
- **Three-level parallelism programming paradigm**
- **Three-task strategy to get there**
- **Tools in the Cray Programming Environment that help**

Future Architectural Directions

- **Nodes are becoming more parallel**
  - More processors per node
  - More threads per processor
  - Vector lengths are getting longer
  - Memory hierarchy is becoming more complex
  - Scalar performance is not increasing and will start decreasing

- **For the next decade, HPC systems will have the same basic architecture that supports:**
  - Message passing between nodes
  - Multithreading within the node (pure MPI will not do)
  - Vectorization at the lowest level (SSE, AVX, GPU, MIC)

Memory Hierarchy is Becoming More Complex

- **As processors get faster, memory bandwidth cannot keep up, resulting in:**
  - More complex caches
  - Non-uniform memory architecture (NUMA) for shared memory on node
  - Operand alignment is becoming more important
- **Going forward: multiple memories within the same address space**
  - Fast expensive memory
  - Slow less expensive memory

Future Application Directions

- **Threading on node as well as vectorization is becoming more important – need more parallelism exploited in applications**

- **Current petascale applications are not structured to take advantage of these architectures**
  - Currently 80-90% of applications use a single level of parallelism
    - message passing between cores of the MPP system
  - Looking forward, application developers are faced with a significant task in preparing their applications for the future
    - Codes must be converted to use multiple levels of parallelism
    - More complex memory hierarchies will require user intervention to achieve good performance

Tools Needed When Creating Hybrid Codes

- **Tools to design your application to be performance-portable across a wide range of systems**
  - Application developers want to develop a single code that can run efficiently on multi-core nodes with or without an accelerator
- **A good Programming Environment that closes the gap between observed performance and achievable performance**
  - A lot more than just a compiler
- **Tools to understand your application**
  - Where is the time spent?
  - Where are the key parallel structures (DO loops)?
  - Where are the important arrays used?
  - What prohibits parallelizing these structures?

WARNING!!!

- **Nothing comes for free, nothing is automatic**
  - Hybridization of an application is difficult
  - Efficient code requires interaction with the compiler to generate
    - High level OpenMP structures
    - Low level vectorization of major computational areas
- **Performance is also dependent upon the location of the data**
  - CPU: NUMA, first-touch
  - Accelerator: resident vs data-sloshing
- **Bottom line: you must understand your application extremely well to achieve good performance on today's and tomorrow's architectures**
- **Software such as Cray's Hybrid Programming Environment provides tools to help, but cannot replace the developer's inside knowledge**

Three Levels of Parallelism Required

1. Developers will continue to use MPI between nodes or sockets

2. Developers must address using a shared memory programming paradigm on the node

3. 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

When to Move to a Hybrid Programming Model

- **When code is network bound**
  - Look at collective time, excluding sync time: this goes up as network becomes a problem
  - Look at point-to-point wait times: if these go up, network may be a problem
- **When MPI starts leveling off**
  - Too much memory used, even if on-node shared communication is available
  - As the number of MPI ranks increases, more off-node communication can result, creating a network injection issue
- **When contention of shared resources increases**
- **When you want to exploit heterogeneous nodes**

Optimization Tips for Multi-core Systems

- Reduce 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

Moving to a Hybrid Code: a Three-Task Approach

1. **Identify potential loops to accelerate**
   - 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. **Parallelize and vectorize identified loops**
   - Split loop work among threads
     - Do parallel analysis and restructuring on targeted high level loops (Reveal loopmark feedback and source browsing)

3. **Add OpenMP and then OpenACC directives**
   - 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

We want a performance-portable application at the end

Task 1: Identify Potential Loops to Accelerate

- **Identify high level computational structures that account for a significant amount of time (95-99%)**
  - GPU maybe 50x faster than core
    - Amdahl's Law:
      - 95% port to GPU with 50x acceleration gives 14x speed-up
      - Compare this with OpenMP over 16 cores...
  - To do this, one must obtain global runtime statistics of the application
    - High level call tree with subroutines and DO loops showing inclusive/exclusive time, min, max, average iteration counts.

- **Tools that will be needed**
  - Advanced instrumentation to measure
    - DO loop statistics, iteration counts, inclusive time to provide estimates for amount of work within a loop
    - Routine level sampling and profiling
  - Cray performance tools offer this

Loop Profile

- **Helps identify high-level serial loops to parallelize**
  - Based on runtime analysis, approximates how much work exists within a loop
  - Provides min, max and average trip counts that can be used to approximate work and help carve up loop on GPU

Loop Work Estimates Report

```
Table 2:  Loop Stats by Function (from -hprofile_generate)
     Loop   |    Loop   |    Loop   |   Loop   |   Loop   |Function=/.LOOP[.]
     Incl   |     Hit   |   Trips   |  Trips   |  Trips   |
     Time   |           |     Avg   |    Min   |    Max   |
    Total   |           |           |          |          |
|-------------------------------------------------------------------------
| 12.697994 |        1 |       20 |      20 |      20 |mg_.LOOP.4.li.253
|  7.510926 |      191 |   86.901 |       2 |     256 |resid_.LOOP.1.li.615
|  7.510130 |    16598 |  201.368 |       2 |     256 |resid_.LOOP.2.li.616
|  4.398111 |  3342308 |  237.924 |       4 |     258 |resid_.LOOP.3.li.617
|  3.557487 |      189 |   56.778 |       1 |     256 |psinv_.LOOP.1.li.544
|  3.557049 |    10731 |      171 |       1 |     256 |psinv_.LOOP.2.li.545
|  2.906496 |  3342308 |  235.924 |       2 |     256 |resid_.LOOP.4.li.623
|  2.133298 |  1835001 |  221.429 |       3 |     258 |psinv_.LOOP.3.li.546
|  1.325157 |  1835001 |  219.429 |       1 |     256 |psinv_.LOOP.4.li.552
|  1.305171 |       21 |         8 |       8 |       8 |mg3p_.LOOP.1.li.473
|  1.291856 |      168 |   31.875 |       1 |     128 |rprj3_.LOOP.1.li.703
|  1.291572 |     5355 |   85.667 |       1 |     128 |rprj3_.LOOP.2.li.705
|  1.137762 |      168 |   32.875 |       2 |     129 |interp_.LOOP.01.li.784
|  1.137482 |     5523 |   85.030 |       2 |     129 |interp_.LOOP.02.li.785
```

subroutine

line number

nested loops
- Loop Hits multiply
- Incl Times reduce

Task 2: Parallelize and Vectorize Identified Loops

- **1st level of parallelism**
  - MPI already there
  - previously used between cores, now between nodes
- **Now introduce two further levels:**
- **2nd level of parallelism**
  - Targeting either:
  - CPU: OpenMP threads within a node
  - GPU: threadblocks, warps within a threadblock
- **3rd level of parallelism**
  - Targeting either:
  - CPU: Vector instructions (SSE, AVX...)
  - GPU: threads within a warp

Challenges

- **Investigate parallelizability of high level looping structures**
  - Often times one level of loop is not enough, must have several parallel loops
    - Need a large number of loop iterations to feed the GPU threads
  - User must understand which high level DO loops have independent iterations
  - Without tools, variable scoping of high level loops is very difficult
    - Loops must be more than independent, their variable usage must adhere to private data local to a thread or global shared across all the threads
    - Independence can be complicated to understand (and even runtime dependent)
- **Investigate vectorizability of lower level DO loops**

Compiler Feedback and Cray Reveal

- **Cray PE can supply information to help**
  - Compiler messages
    - written to STDOUT
    - 5 levels of severity (ERROR down to COMMENT)
    - Compiler flag controls level of messages reported
  - Loopmark
    - optional: `cc -rm -c my_program.c`
    - written in *.lst files
    - annotated listing file with information on loop scheduling
    - very useful for identifying vectorisation opportunities
      - can then tune behaviour using Cray directives (see `man intro_directives`)
  - Program library
    - New feature
    - Allows whole program analysis for more in-depth inter-procedural analysis
      - Important when calltree spans more than one source file
    - Allows loop variables to be scoped
    - `cc -hwp -h pl=/path_to_my_program_library/ <other options>`
- **Lots of information, displayed in lots of places**
  - Cray's Reveal tool pulls this together
  - Use it to view loopmark information, compiler messages, browse source

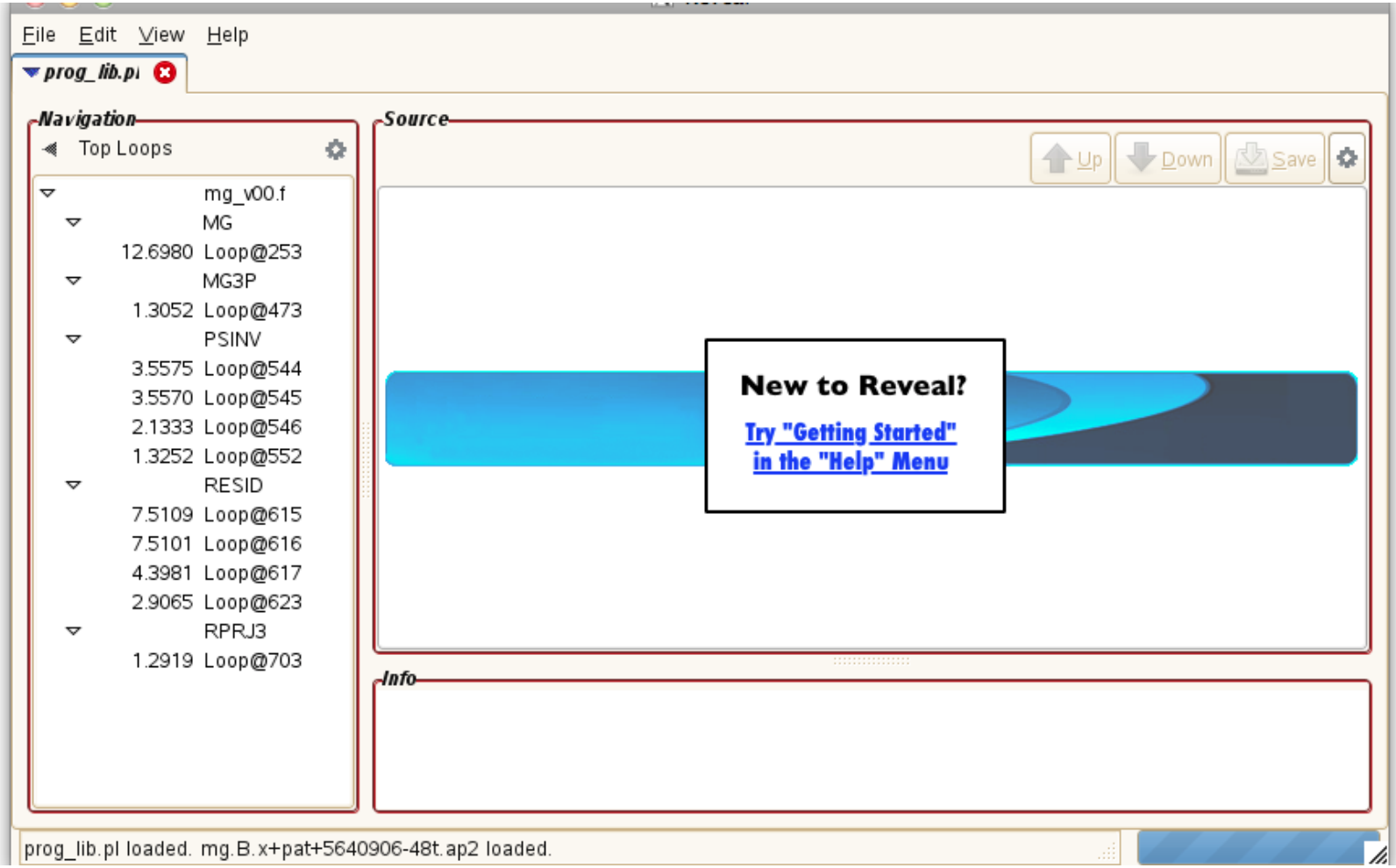
Reveal

New code analysis and restructuring assistant…

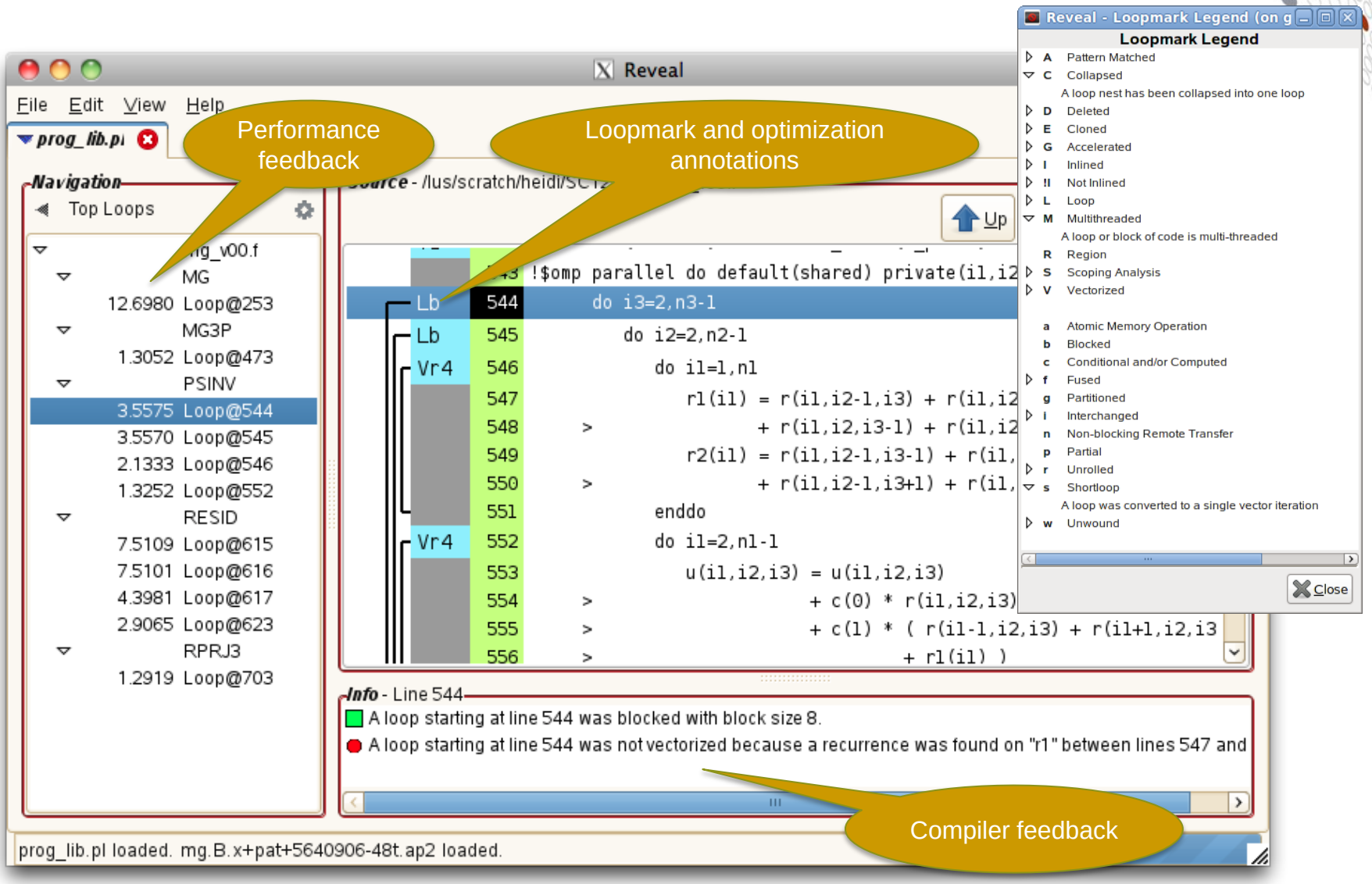
- **Uses both the performance toolset and CCE's program library functionality to provide static and runtime analysis information**

- **Key Features**
  - Annotated source code with compiler optimization information
    - Feedback on critical dependencies that prevent optimizations
  - Scoping analysis
    - Identify, shared, private and ambiguous arrays
      - Allow user to privatize ambiguous arrays
      - Allow user to override dependency analysis
  - Source code navigation based on performance data collected through CrayPat

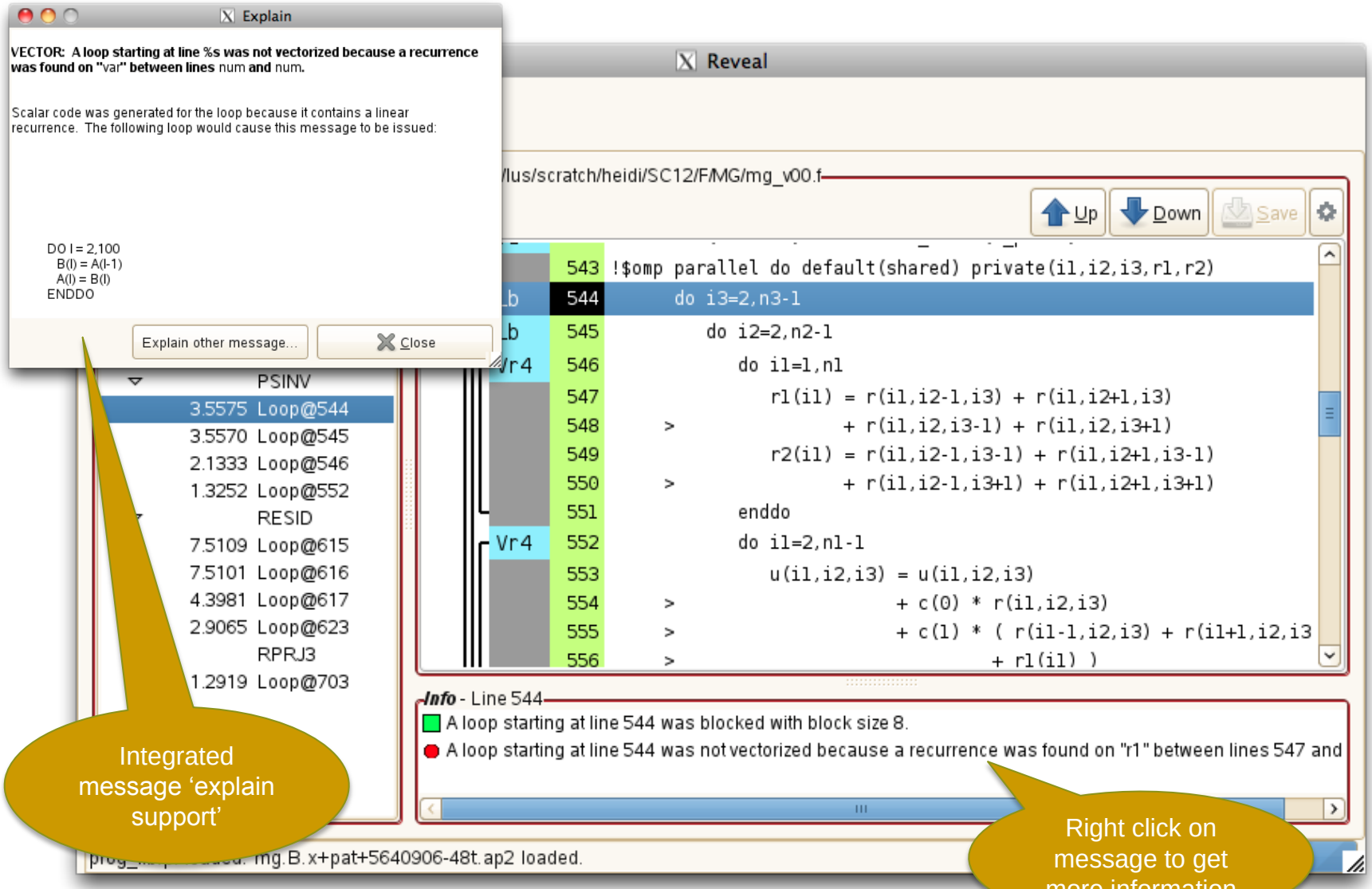
Navigate Source Based on Important Loops



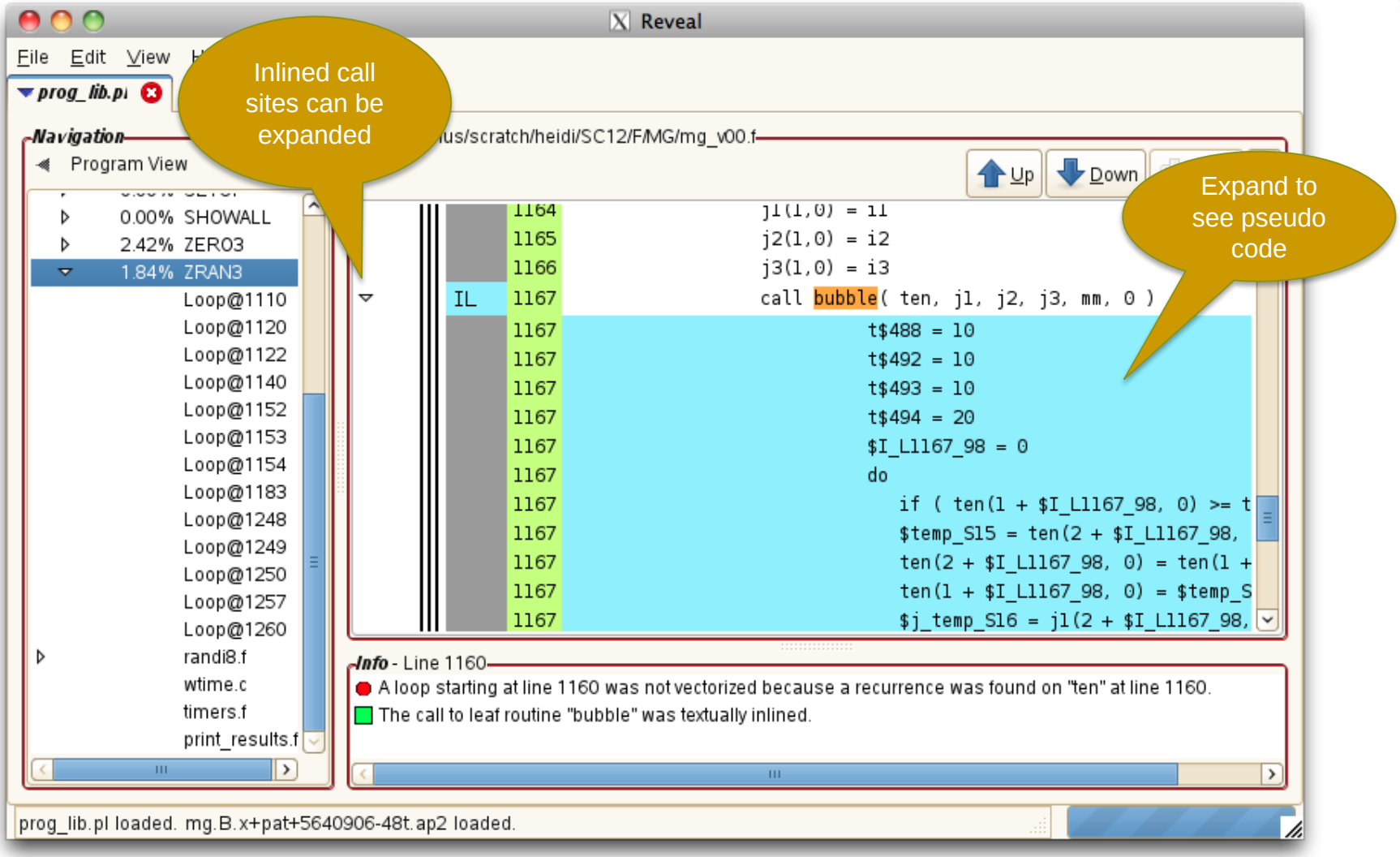
Review Loopmark with Performance Information



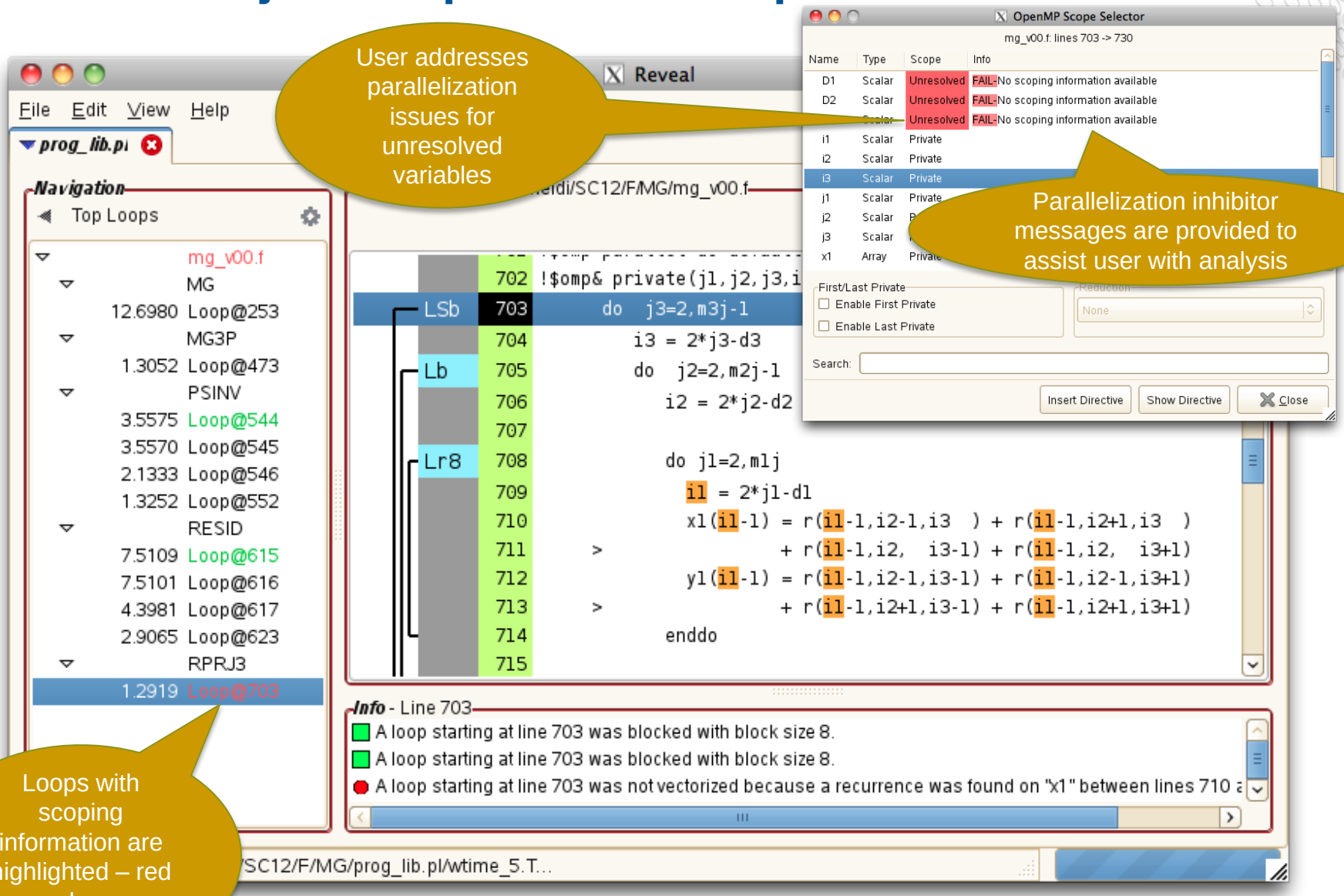
View More Information About an Optimization



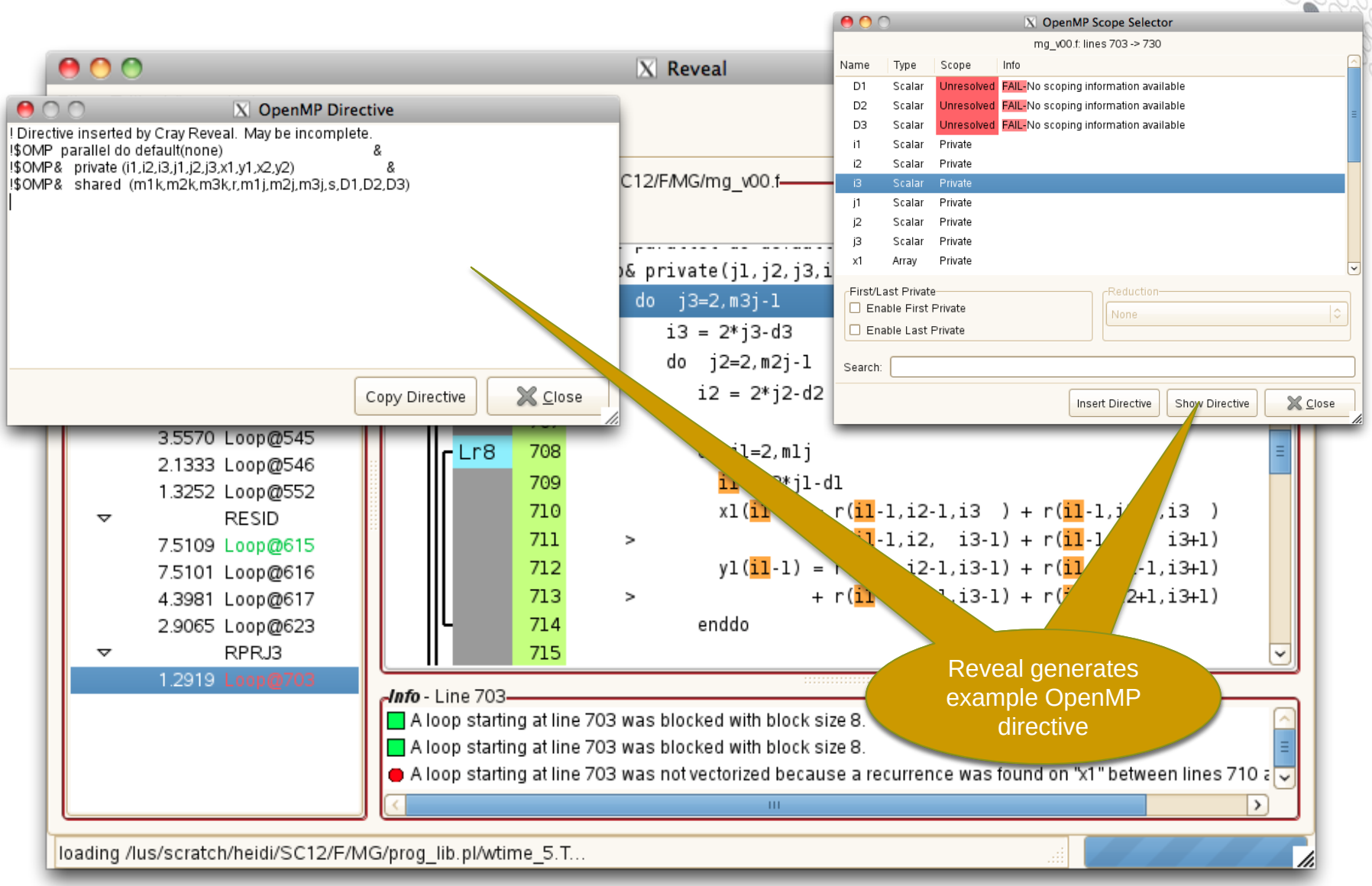
Review Inlined Functions Within Loops



Parallelize Major Computational Loops



Task 3: Add OpenMP Directives



Introduce OpenACC

- Profile the hybrid MPI/OpenMP code to ensure it performs well.
- Add OpenACC directives once a very good MPI/OpenMP hybrid code is in hand

Summary (1)

- **Hybridizing a code gives many performance advantages**
  - Resource contention (network, node memory...)
  - Things will only get worse in the future
    - Foreseeable HPC architectures, not just the EU economies
- **Users should look for three-level parallelism**
  - Message passing between nodes (MPI, CAF...)
  - OpenMP shared memory within the node (or NUMA node)
  - Vectorization within the core (SSE, AVX, accelerator)
- **Getting to OpenACC via OpenMP is a good idea**
  - Same work required
  - Can have both (conditionally compile one or other or none)
  - First level of debugging on multicore CPU

Summary (2)

- **Three-task strategy to get there**
  1. Identification of possible accelerator kernels
  2. Parallel analysis, scoping and vectorization
  3. Moving to OpenMP and then to OpenACC

- **The Cray Programming Environment has tools to help**
  - CrayPAT (use with any compiler)
  - Cray Compiler
  - Reveal

- **Vendor lock-in?**
  - OpenMP, OpenACC are Open Standards and vendor-agnostic
  - Directives are plain text
    - The compiler doesn't know or care how you created them

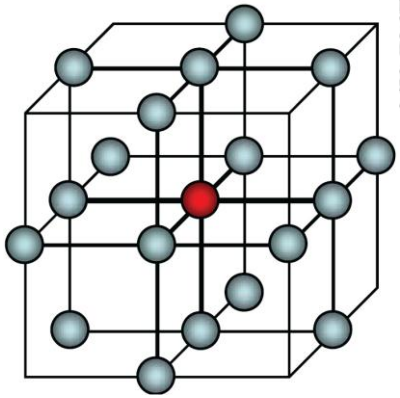
Porting a Simple Code

Worked example: scalar Himeno code

Overview

- **This worked example leads you through accelerating a simple application**
  - a simple application is easy to understand
  - but it shows all the steps you would use for a more complicated code
- **In the following practical, you will work through these steps yourself**

The Himeno Benchmark



- 3D Poisson equation solver
  - Iterative loop evaluating 19-point stencil
  - Memory intensive, memory bandwidth bound
- Fortran and C implementations are available from http://accc.riken.jp/2444.htm
- We look here at the scalar version for simplicity
  - We will discuss the parallel version later today
- Code characteristics
  - Around 230 lines of Fortran or C
  - Arrays statically allocated
    - problem size fixed at compile time

Why use such a simple code?

- **Understanding a code structure is crucial if we are to *successfully* OpenACC an application**
  - i.e. one that runs faster node-for-node
  - not just full accelerator vs. single CPU core
- **There are two key things to understand about the code:**
  - How is data passed through the calltree?
  - Where are the hotspots?
- **Answering these questions for a large application is hard**
  - There are tools to help
    - we will discuss some of them later in the tutorial
  - With a simple code, we can do all of this just by code inspection

The key questions in detail

- **How is data passed through the calltree?**
  - CPUs and accelerators have separate memory spaces
  - The PCIe link between them is relatively slow
  - Unnecessary data transfers will wipe out any performance gains
  - A successful OpenACC port will keep data resident on the accelerator
- **Where are the hotspots?**
  - The OpenACC programming model is aimed at loop-based codes
    - Which loopnests dominate the runtime?
    - Are they suitable for an accelerator?
      - What are the min/average/max tripcounts?
- **Minimising data movements will probably require acceleration of many more (and possibly all) loopnests**
  - Not just the hotspots
    - any loopnest that processes arrays that we want accelerator-resident
  - But we have to start somewhere

Stages to accelerating an application

1. **Understand and characterise the application**
   - Profiling tools, code inspection, speaking to developers if you can
2. **Introduce first OpenACC kernels**
3. **Introduce data regions in subprograms**
   - reduce unnecessary data movements
   - will probably require more OpenACC kernels
4. **Move up the calltree, adding higher-level data regions**
   - ideally, port entire application so data arrays live entirely on the GPU
   - otherwise, minimise traffic between CPU and GPU
   - This will give the single biggest performance gain
5. **Only now think about performance tuning for kernels**
   - First correct any obviously inefficient scheduling on the GPU
     - This will give some good performance improvements
   - Optionally, experiment with OpenACC tuning clauses
     - You may gain some final additional performance from this

- **And remember Amdahl's law...**

Step 1: Himeno program structure

- **Code has two subprograms**
  - init_mt() initialises the data array
    - Called once at the start of the program
  - jacobi() performs iterative stencil updates of the data array
    - The number of updates is an argument to the subroutine and fixed
      - A summed residual is calculated, but not tested for convergence
    - This subroutine is called twice:
      - Each call is timed internally by the code
      - The first call does a calibration run
        - small, fixed number of iterations.
        - The time is used to estimate how many iterations could be done in one minute
      - The second call does a measurement run
        - computed number of iterations
        - The time is converted into a performance figure by the code

        - Actually, it is useful when testing to do a fixed number of iterations
        - Then we can use the value of the residual for a correctness check.

  - The next slide shows an edited version of the code
    - These slides discuss the Fortran version; there is also a C code

Step 1: Himeno program structure (contd)

calibration run

measurement run

```
PROGRAM himeno       ! EDITED HIGHLIGHTS ONLY
   CALL initmt       ! Initialise local matrices

   cpu0 = gettime() ! Wraps SYSTEM_CLOCK
   CALL jacobi(3,gosa)
   cpu1 = gettime()
   cpu = cpu1 - cpu0

!  nn = INT(ttarget/(cpu/3.0)) ! Fixed runtime
   nn = 1000         ! Hardwired for testing

   cpu0 = gettime()
   CALL jacobi(nn,gosa)
   cpu1 = gettime()
   cpu = cpu1 - cpu0

   xmflops2 = flop*1.0d-6/cpu*nn

   PRINT *,' Time:   ',cpu
   PRINT *,' Gosa:   ',gosa
   PRINT *,' MFLOPS:',xmflops2
END PROGRAM himeno
```

- Now we look at the details of jacobi()

Step 1: Structure of the jacobi routine

- **Iteration loop**
  - must be sequential!
- **Apply stencil to `p`**
  - create temporary **`wrk2`**
  - residual **`gosa`** computed
    - details on the next slide
- **Update array `p`**
  - from **`wrk2`**
  - can be parallelised
  - outer halo unchanged

```
SUBROUTINE jacobi(nn,gosa)

  iter_lp: DO loop = 1,nn

! compute stencil: wrk2, gosa from p
     <described on next slide>

! copy back wrk2 into p
        DO k = 2,kmax-1
           DO j = 2,jmax-1
              DO i = 2,imax-1
                 p(i,j,k) = wrk2(i,j,k)
              ENDDO
           ENDDO
        ENDDO

   ENDDO iter_lp

END SUBROUTINE jacobi
```

Step 1: The Jacobi computational kernel

- Stencil applied to array **p**
  - 19-point finite difference
- Not in-place:
  - Updated values saved in temporary array **wrk2**
- Residual value **gosa**
  - computed here
  - is a reduction variable
- This loopnest dominates runtime
  - Can be parallelised

```
gosa = 0d0
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)
   gosa = gosa + ss*ss
   wrk2(i,j,k) = p(i,j,k) + omega*ss
  ENDDO
 ENDDO
ENDDO
```

fwd n.n.
n.n.n.
bwd n.n.

Step 2: a first OpenACC kernel

- Start with most expensive
  - apply parallel loop
  - end parallel loop optional
    - *advice: use it for clarity*
- reduction clause
  - like OpenMP, not optional
- private clause
  - By default:
    - loop variables private
      - like OpenMP
    - scalar variables private
      - ***not*** like OpenMP
  - so clause optional here
    - *advice: use it for clarity*
  - N.B. private arrays
    - always need private clause

```
gosa = 0d0

!$acc parallel loop reduction(+:gosa) &
!$acc   private(i,j,k,so,ss)


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 ) &
     <etc...>

   ss = (s0*a(i,j,k,4) - p(i,j,k)) * &
                                 bnd(i,j,k)
   gosa = gosa + ss*ss
   wrk2(i,j,k) = p(i,j,k) + omega*ss
  ENDDO
 ENDDO
ENDDO
!$acc end parallel loop
```

Step 2: a first OpenACC kernel (contd)

- copy* data clauses
  - compiler will do automatic analysis
    - usually correct
    - but can be over-cautious
- ***advice:***
  - *only use clauses if compiler over-cautious*
  - explicit data clauses will interfere with data directives at next step

```
gosa = 0d0

!$acc parallel loop reduction(+:gosa) &
!$acc&  private(i,j,k,so,ss) &
!$acc&  copyin(p,a,b,c,bnd,wrk1) &
!$acc&  copyout(wrk2)
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 ) &
     <etc...>

   ss = (s0*a(i,j,k,4) - p(i,j,k)) * &
                              bnd(i,j,k)
   gosa = gosa + ss*ss
   wrk2(i,j,k) = p(i,j,k) + omega*ss
  ENDDO
 ENDDO
ENDDO
!$acc end parallel loop
```

Compiler feedback

- **Compiler feedback is extremely important**
  - Did the compiler recognise the accelerator directives?
    - A good sanity check
  - How will the compiler move data?
    - Only use data clauses if the compiler is over-cautious on the copy*
    - Or you want to declare an array to be scratch space (create clause)
    - First major code optimisation: removing unnecessary data movements
  - How will the compiler schedule loop iterations across GPU threads?
    - Did it parallelise the loopnests?
    - Did it schedule the loops sensibly?
    - The other main optimisation is correcting obviously-poor loop scheduling
- **Compiler teams work very hard to make feedback useful**
  - *advice: use it, it's free!* (i.e. no impact on performance to generate it)
    - CCE: -hlist=a Produces commentary files <stem>.lst
    - PGI: -Minfo Feedback to STDERR

How a kernel is executed on a GPU

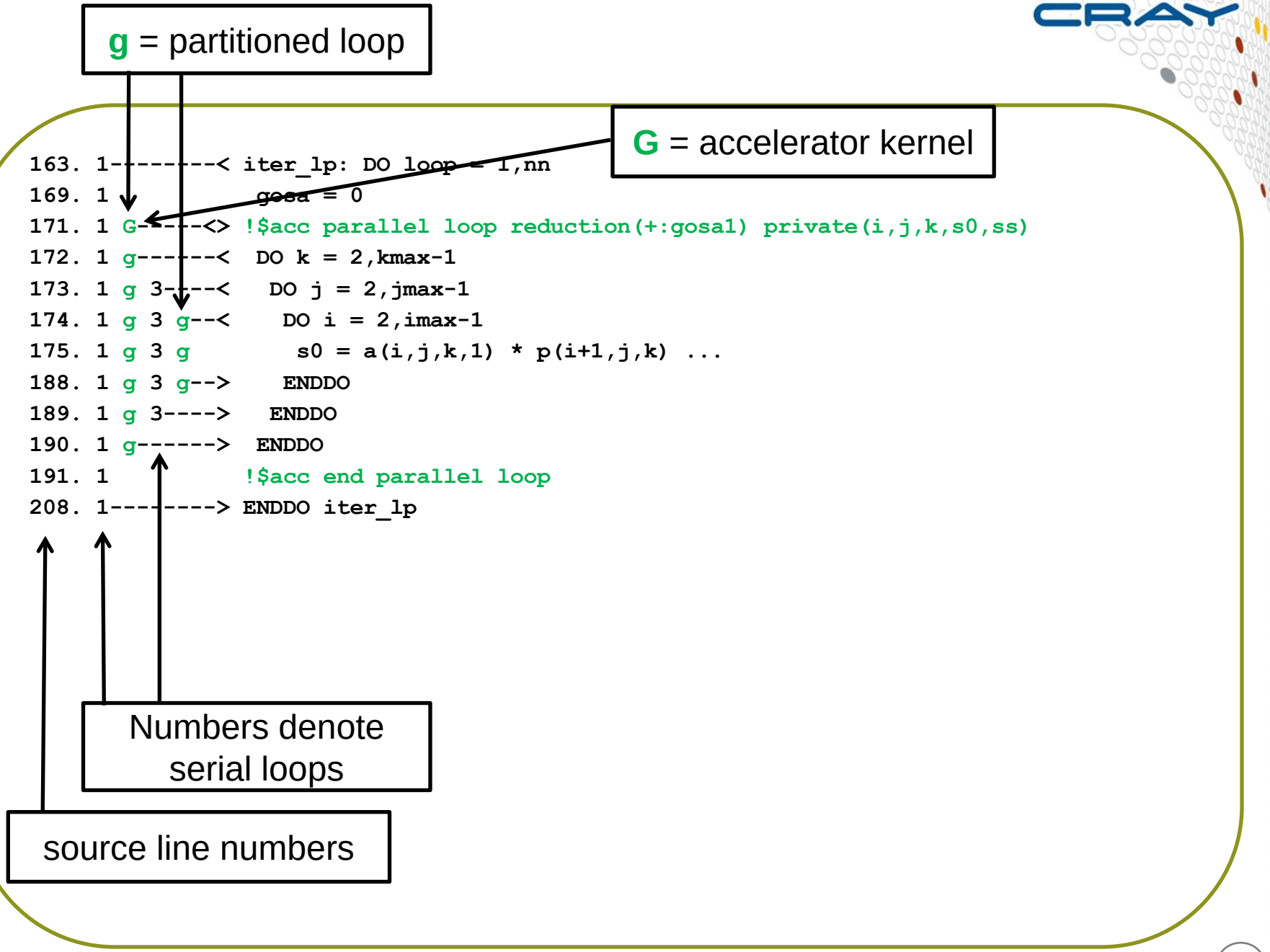
- **A GPU kernel is executed by a large pool of threads**
  - Whether we use OpenACC or CUDA
  - The threads are divided into sets called threadblocks
  - Each threadblock executes on a different piece of hardware
    - Symmetric Multiprocessor (SM)
    - (Almost) like a single vector processor

- **Within a threadblock**
  - The threads are logically divided into sets called warps
    - threads within a warp (32 threads) are executed concurrently
    - (almost) like a set of vector instructions, each of fixed width 32
  - so the threadblock executes as a sequence of concurrent warps

- **The CUDA programming model**
  - does not make warps explicit
  - but you need to know about them to understand code performance

OpenACC scheduling

- **OpenACC makes the distinction explicit**
  - gang refers to threadblocks (just as in CUDA)
  - worker refers to entire warps
    - i.e. entire width-32 vector instructions (32 is fixed by the hardware)
  - vector refers to threads within a warp
    - i.e. 32 threads that do the same thing at the same time
- **When a loopnest is partitioned by the compiler**
  - The loop iterations are divided up into gangs of workers
    - each worker executes sets of vector instructions
  - There are many different ways for the compiler to do this
    - If you give no further instruction, the compiler will make a decision
    - You can see what this was from the compiler feedback
    - You can then over-ride this decision (at the next compilation)
      - by adding additional OpenACC directives and clauses
- **This is where OpenACC differs from OpenMP**
  - OpenMP parallel regions only partition one loop over the CPU threads
  - OpenACC parallel/kernels regions partition up to three nested loops



```
163. 1--------< DO loop = 1,nn
169. 1           gosa1 = 0
171. 1 G-----<> !$acc parallel loop reduction(+:gosa1) private(i,j,k,s0,ss)
172. 1 g------<  DO k = 2,kmax-1
173. 1 g 3----<   DO j = 2,jmax-1
174. 1 g 3 g--<    DO i = 2,imax-1
175. 1 g 3 g        s0 = a(i,j,k,1) * p(i+1,j,k) ...
188. 1 g 3 g-->    ENDDO
189. 1 g 3---->   ENDDO
190. 1 g------>  ENDDO
191. 1          !$acc end parallel loop
208. 1--------> ENDDO

Data movements:
ftn-6418 ftn: ACCEL File = himeno_F_v02.F90, Line = 171
  If not already present: allocate memory and copy whole array "p" to accelerator,
   free at line 191 (acc_copyin).

<identical messages for a,b,c,wrk1,bnd>

ftn-6416 ftn: ACCEL File = himeno_F_v02.F90, Line = 171
  If not already present: allocate memory and copy whole array "wrk2" to accelerator,
   copy back at line 191 (acc_copy).
```

To learn more, use command:
explain ftn-6418

yes, as we expected

Over-cautious: compiler worried about halos;
could specify copyout(wrk2)

```
163. 1--------< DO loop = 1,nn
169. 1           gosa1 = 0
171. 1 G-----<> !$acc parallel loop reduction(+:gosa1) private(i,j,k,s0,ss)
172. 1 g------<  DO k = 2,kmax-1
173. 1 g 3----<   DO j = 2,jmax-1
174. 1 g 3 g--<    DO i = 2,imax-1
175. 1 g 3 g        s0 = a(i,j,k,1) * p(i+1,j,k)
188. 1 g 3 g-->    ENDDO
189. 1 g 3---->   ENDDO
190. 1 g------>  ENDDO
191. 1          !$acc end parallel loop
208. 1--------> ENDDO

ftn-6430 ftn: ACCEL File = himeno_F_v02.F90, Line = 172
  A loop starting at line 172 was partitioned across the thread blocks.

ftn-6509 ftn: ACCEL File = himeno_F_v02.F90, Line = 173
  A loop starting at line 173 was not partitioned because a better candidate was found at
    line 174.

ftn-6412 ftn: ACCEL File = himeno_F_v02.F90, Line = 173
  A loop starting at line 173 will be redundantly executed.

ftn-6430 ftn: ACCEL File = himeno_F_v02.F90, Line = 174
  A loop starting at line 174 was partitioned across the 128 threads within a threadblock.
```

gangs, like **CUDA** threadblocks:
k value(s) built from blockidx.x

Each thread executes complete
complete j-loop for its i, k value(s)

vectors, like **CUDA**:
i value(s) built from threadidx.x

Is the code still correct?

- **Most important thing is that the code is correct:**
  - Make sure you check the residual (`gosa`)
  - N.B. will never get bitwise reproducibility between CPU and GPU architectures
    - different compilers are also likely to give different results

- ***Advice: make sure you have correctness checks***
  - *e.g. checksums, residuals, representative array values*
  - *try to use double precision for checksums, residuals*
    - *even if the code is single precision*
    - *at least for the global sums or other reduction variables*

How does this first version perform?

language	Fortran		C	
precision	single	double	single	double
v00	2881	1454	2287	1131
v01	1177	565	1178	594

- **The code is faster...**
  - ... but not by much and only compared to one CPU core.
- **Why?**
  - Because we are spending most of our time on data transfers
  - Profiling shows only 2% of the GPU time is actual compute
- ***Lesson: optimise data movements before looking at kernel performance***
  - We are lucky with Himeno
  - most codes are actually slower than one core at this stage

Profiling the first Himeno kernel

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

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

 100.0% | 11.716 | 11.656 |     23525 |      1680 |     515 |Total
|-------------------------------------------------------------------------------------------
| 100.0% | 11.716 | 11.656 |     23525 |      1680 |     515 |main_
|        |        |        |           |           |         | jacobi_
3        |        |        |           |           |         |  jacobi_.ACC_REGION@li.288
||||-----------------------------------------------------------------------------------------
4|||  93.5% | 10.953 | 10.911 |     23525 |        -- |     103 |jacobi_.ACC_COPY@li.288
4|||   4.5% |  0.527 |  0.517 |        -- |      1680 |     103 |jacobi_.ACC_COPY@li.315
4|||   2.0% |  0.230 |     -- |        -- |        -- |     103 |jacobi_.ACC_SYNC_WAIT@li.315
4|||   0.0% |  0.004 |  0.228 |        -- |        -- |     103 |jacobi_.ACC_KERNEL@li.288
4|||   0.0% |  0.001 |     -- |        -- |        -- |     103 |jacobi_.ACC_REGION@li.288(exclusive)
|=========================================================================================
```

- **CrayPAT profile, breaks time down into compute and data**
- **Most kernels are launched asynchronously**
  - as is the case with CUDA
  - reported host time is the time taken to launch operation
    - Host time is much smaller than accelerator time
  - Host eventually waits for completion of accelerator operations
    - This shows up in a "large" SYNC_WAIT time

Profiling the first Himeno kernel

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

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

 100.0% | 11.745 | 11.686 |     23525 |      1680 |     412 |Total
|-------------------------------------------------------------------------------------------
| 100.0% | 11.745 | 11.686 |     23525 |      1680 |     412 |main_
|        |        |        |           |           |         | jacobi_
3        |        |        |           |           |         |  jacobi_.ACC_REGION@li.288
||||-----------------------------------------------------------------------------------------
4|||  93.5% | 10.978 | 10.935 |     23525 |        -- |     103 |jacobi_.ACC_COPY@li.288
4|||   4.5% |  0.532 |  0.523 |        -- |      1680 |     103 |jacobi_.ACC_COPY@li.315
4|||   2.0% |  0.234 |  0.228 |        -- |        -- |     103 |jacobi_.ACC_KERNEL@li.288
4|||   0.0% |  0.001 |     -- |        -- |        -- |     103 |jacobi_.ACC_REGION@li.288(exclusive)
|=========================================================================================
```

- **Clarify profile by inserting synchronisation points**
  - Could do this explicitly by inserting "acc wait" after every operation
  - better to compile with CCE using -hacc_model=auto_async_none
    - see man crayftn for details
- **Profile now shows same time for host at every operation**
  - It is now very clear that data transfers take most of the time
- **Extra synchronisation will affect performance**
  - Could skew the profile, so use with care
  - N.B. GPU profilers (Craypat, Nvidia...) already introduce some sync.

Step 3: Optimising data movements

- **Within jacobi routine**
  - data-sloshing: all arrays copied to/from GPU at every loop iteration
- **Need to establish data region outside the iteration loop**
  - Then data can remain resident on GPU for entire call
    - reused for each iteration without copying to/from host
- **Must accelerate all loopnests processing the arrays**
  - Even if they takes negligible compute time,
  - must still accelerate for data locality
    - This can be a lot of work
      - Performance of the kernels is irrelevant
    - A major productivity win for OpenACC compared to low-level languages
      - You can accelerate a loopnest with one directive; usually no need for tuning
      - You don't have to handcode a new CUDA kernel

Step 3: Structure of the jacobi routine

- data region spans iteration loop
  - includes both CPU and accelerator code
  - need explicit data clauses
    - no automatic scoping
    - requires knowledge of app
  - enclosed kernels
    - if in same routine
      - no data clauses for these variables
    - if in different routine
      - use present clause for these variables
    - see earlier advice
  - **wrk2** now a scratch array
    - does not need copying

```
SUBROUTINE jacobi(nn,gosa)

!$acc data copy(p) &
!$acc&     copyin(a,b,c,wrk1,bnd) &
!$acc&     create(wrk2)
   iter_lp: DO loop = 1,nn

      gosa = 0d0
! compute stencil: wrk2, gosa from p
!$acc parallel loop <clauses>
      <stencil loopnest>
!$acc end parallel loop

! copy back wrk2 into p
!$acc parallel loop
      <copy loopnest>
!$acc end parallel loop

   ENDDO iter_lp
!$acc end data

END SUBROUTINE jacobi
```

How does this second version perform?

language	Fortran		C	
precision	single	double	single	double
v00	2881	1454	2287	1131
v01	1177	565	1178	594
v02	37525	20300	37143	20287

- **A big performance improvement**
  - Now 51% of the GPU time is compute
    - And more of the profile has been ported to the GPU
  - Data transfers only happen once per call to jacobi(),
    - rather than once per iteration
  - Code still correct:
    - Check the `gosa` values

Profile with a local data region in jacobi()

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

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

 100.0% | 0.497 | 0.475 |   424.177 |    32.630 |     624 |Total
|-------------------------------------------------------------------------------------
| 100.0% | 0.497 | 0.475 |   424.177 |    32.630 |     624 |main_
|        |       |       |           |           |         | jacobi_
3        |       |       |           |           |         |  jacobi_.ACC_DATA_REGION@li.276
||||-----------------------------------------------------------------------------------
4|||  50.5% | 0.251 | 0.236 |     0.001 |     0.001 |     412 |jacobi_.ACC_REGION@li.288
|||||---------------------------------------------------------------------------------
5||||  46.7% | 0.232 | 0.227 |        -- |        -- |     103 |jacobi_.ACC_KERNEL@li.288
5||||   1.9% | 0.010 | 0.005 |        -- |     0.001 |     103 |jacobi_.ACC_COPY@li.315
5||||   1.8% | 0.009 | 0.004 |     0.001 |        -- |     103 |jacobi_.ACC_COPY@li.288
|||||=================================================================================
4|||  40.0% | 0.199 | 0.197 |   424.176 |        -- |       2 |jacobi_.ACC_COPY@li.276
4|||   7.6% | 0.038 | 0.033 |        -- |        -- |     206 |jacobi_.ACC_REGION@li.317
5|||   7.5% | 0.037 | 0.033 |        -- |        -- |     103 | jacobi_.ACC_KERNEL@li.317
4|||   1.9% | 0.009 | 0.009 |        -- |    32.629 |       2 |jacobi_.ACC_COPY@li.335
|=====================================================================================
```

- **Profile now dominated by compute (ACC_KERNEL)**
- **Data transfers infrequent**
  - only once for each of 2 calls to jacobi
  - but still very expensive

Step 4: Further optimising data movements

- **Still including single copy of data arrays in timing of jacobi routine**

- **Solution: move up the call tree to parent routine**
  - Add data region that spans both initialisation and iteration routines
  - Specified arrays then only move on boundaries of outer data region
    - moves the data copies outside of the timed region
      - after all, benchmark aims to measure flops, not PCIe bandwidth

Adding a data region

- Spans both calls to jacobi
  - plus timing calls
- Arrays just copyin now
  - and transfers not timed
  - **wrk2** could be create
- Keep data region in jacobi
  - you can nest data regions
  - arrays now declared present on inner region
  - could be copy_or_present
    - *advice: use present*
    - runtime error if not present
    - rather than just wrong result
- Drawback: arrays have to be in scope for this to work
  - may need to unpick clever use of module data
  - or use OpenACC v2.0 unstructured data regions

```
PROGRAM himeno
   CALL initmt

!$acc data copyin(p,a,b,c,bnd,wrk1,wrk2)
   cpu0 = gettime()
   CALL jacobi(3,gosa)
   cpu1 = gettime()

   cpu0 = gettime()
   CALL jacobi(nn,gosa)
   cpu1 = gettime()
!$acc end data

   END PROGRAM himeno
```

```
SUBROUTINE jacobi(nn,gosa)

!$acc data present(p,a,b,c,wrk1,bnd,wrk2)
   iter_lp: DO loop = 1,nn

   ENDDO iter_lp
!$acc end data

END SUBROUTINE jacobi
```

Step 4: Going further

- **Best solution is to port entire application to GPU**
  - data regions span entire use of arrays
  - all enclosed loopnests accelerated with OpenACC
  - no significant data transfers

- **Expand outer data region to include call to initialisation routine**
  - arrays can now all be declared as scratch space with create clause
  - need to accelerated loopnests in initmt(), declaring arrays present

- **N.B. No easy way to ONLY allocate arrays in GPU memory**
  - CPU version is now dead space, but
  - GPU memory is usually the limiting factor, so usually not a problem
  - Can use OpenACC API calls to do this, if really want to
    - Means more OpenACC-specific code changes

Porting entire application

- No significant data transfers now
  - doesn't improve measured performance in this case

```
PROGRAM himeno

!$acc data create(p,a,b,c,bnd,wrk1,wrk2)
   CALL initmt

   CALL jacobi(3,gosa)  ! plus timing calls

   CALL jacobi(nn,gosa) ! plus timing calls

!$acc end data

   END PROGRAM himeno
```

```
SUBROUTINE initmt
!$acc data present(p,a,b,c,wrk1,bnd)
!$acc parallel loop
   <set all elements to zero>

!$acc parallel loop
   <set some elements to be non-zero>
!$acc end data

END SUBROUTINE initmt
```

How does this third version perform?

language	Fortran		C	
precision	single	double	single	double
v00	2881	1454	2287	1131
v01	1177	565	1178	594
v02	37525	20300	37143	20287
v03	51921	28863	51078	28891

- **Code is now a lot faster (44x faster than v01)**
  - 98% of the GPU time is now compute
    - Remaining data transfers are negligible and outside region timed
  - And the code is still correct:
    - Check the Gosa values!
- **We're getting a great speedup: 18x compared to v00**
  - But this is compared to one CPU core out of 16
  - What happens if we use all the cores
    - using OpenMP, as this is originally a scalar code

Profile of fully ported application

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

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

 100.0% | 0.296 | 0.275 |     0.001 |     0.001 |     634 |Total
|-------------------------------------------------------------------------------
| 100.0% | 0.296 | 0.275 |     0.001 |     0.001 |     634 |main_
|        |       |       |           |           |         | main_.ACC_DATA_REGION@li.116
|||-----------------------------------------------------------------------------
3||  97.6% | 0.289 | 0.269 |     0.001 |     0.001 |     624 |jacobi_
4||        |       |       |           |           |         | jacobi_.ACC_DATA_REGION@li.277
|||||---------------------------------------------------------------------------
5||||  84.8% | 0.251 | 0.236 |     0.001 |     0.001 |     412 |jacobi_.ACC_REGION@li.288
||||||--------------------------------------------------------------------------
6|||||  78.4% | 0.232 | 0.227 |        -- |        -- |     103 |jacobi_.ACC_KERNEL@li.288
6|||||   3.3% | 0.010 | 0.005 |        -- |     0.001 |     103 |jacobi_.ACC_COPY@li.315
6|||||   3.1% | 0.009 | 0.004 |     0.001 |        -- |     103 |jacobi_.ACC_COPY@li.288
||||||==========================================================================
5||||  12.7% | 0.038 | 0.033 |        -- |        -- |     206 |jacobi_.ACC_REGION@li.317
6||||  12.7% | 0.038 | 0.033 |        -- |        -- |     103 | jacobi_.ACC_KERNEL@li.317
|||||==========================================================================
3||   1.8% | 0.005 | 0.005 |        -- |        -- |       7 |initmt_
4||        |       |       |           |           |         | initmt_.ACC_DATA_REGION@li.208
|=============================================================================
```

- **Almost no data transferred**
  - remainder (gosa and a few compiler internals) hard to remove
- **At this point we can start looking at kernel optimisation**

Step 5: Is this a good loop schedule?

```
172. 1 g------<  DO k = 2,kmax-1
173. 1 g 3----<   DO j = 2,jmax-1
174. 1 g 3 g--<    DO i = 2,imax-1
175. 1 g 3 g         s0 = a(i,j,k,1)*p(i+1,j,k) ...
188. 1 g 3 g-->    ENDDO
189. 1 g 3---->   ENDDO
190. 1 g------>  ENDDO
```

- Look at .lst file
  - compile with -hlist=a
- Should see partitioning both between and across threadblocks
  - if not, much of GPU is being wasted
  - Check every accelerated loopnest:
    - Single loop (or fully collapse-d loopnest)
      - Should see one set of **g** loopmark symbols
    - Multiple loops
      - Should see two (or perhaps three) loops with **g**
    - Should also seeing accompanying messages saying loop was partitioned:
      - "across threadblocks"
      - "across the threads within a threadblock

Step 5: Which loop was vectorised?

- Usually want inner loop to be vectorised
  - i.e. "partitioned across the threads within a threadblock"
  - (assuming it indexes stride-1 index of main data arrays)
  - allows coalesced loading of data to/from global memory

- If inner loop is not vectorised:
  - Is it vectorisable; are loop iterations independent?
    - Yes? Then you need to help the compiler:
      - Put "acc loop vector" directive above the loop statement
      - For C, use `restrict` keyword (or CCE compile flag -hrestrict)
    - No? You need to rewrite the code (it will probably go faster on the CPU)
      - Can you use a more-explicitly parallel algorithm?
        - removing dependencies between loop iterations
      - Avoid incremented counters (e.g. when packing buffers)
      - Change data layout so inner loop addresses fastest-moving array index

- Often the most important optimisation (after data-sloshing)
  - almost guaranteed to give big performance increase
  - other optimisations are often trial-and-error
    - we'll discuss these later in the tutorial

OpenACC and/or OpenMP

- **So far all performance comparisons are unfair**
  - full accelerator versus single CPU core
- **Need an OpenMP version to fully exercise CPU**
- **OpenMP and OpenACC can coexist in same source code**
  1. As alternative for compilation
     - Use compiler flags to enable/disable: -hnoomp, -hnoacc for CCE
  2. On separate code regions
  3. You can nest OpenACC inside OpenMP (but not other way round)
     - Cray OpenACC runtime is OpenMP-threadsafe
     - But beware of too much data movement
  - If you want some routines to be always OpenMP, but others to be sometimes OpenMP and sometimes OpenACC
    - you'll need to preprocess using the _OPENMP and _OPENACC macros

OpenMP then OpenACC

- **Existing OpenMP is a good starting point for OpenACC**
  - assuming it is good OpenMP
    - correct, with large parallel regions high up the callchain
- **If your code does not have OpenMP**
  - it may make sense to first apply OpenMP and then add OpenACC
    - A lot of the "thinking" work can be reused
    - You can do first-level parallel debugging on the CPU
    - The hybrid MPI/OpenMP CPU code may well perform better

Aside: OpenACC and OpenMP (naïve)

```
!$acc data etc.

   iteration: DO loop = 1, nn


      gosa = 0d0

      gosa1 = 0d0
!$acc parallel loop
      reduction(+:gosa1) etc.
      <stencil loopnest>
!$acc end parallel loop

!$acc parallel loop
      <copy loopnest>
!$acc end parallel loop


      gosa = gosa + gosa1

   ENDDO iteration
!$acc end data
```

```
iteration: DO loop = 1, nn


   gosa = 0d0

   gosa1 = 0d0
!$omp parallel do
      reduction(+:gosa1) etc.
      <stencil loopnest>
!$omp end parallel do

!$omp parallel do etc.
      <copy loopnest>
!$omp end parallel do


   gosa = gosa + gosa1

ENDDO iteration
```

Aside: OpenACC and OpenMP (better)

```
!$acc data etc.

   iteration: DO loop = 1, nn


      gosa = 0d0

      gosa1 = 0d0
!$acc parallel loop
      reduction(+:gosa1) etc.
      <stencil loopnest>
!$acc end parallel loop

!$acc parallel loop
      <copy loopnest>
!$acc end parallel loop


      gosa = gosa + gosa1

   ENDDO iteration
!$acc end data
```

```
!$omp parallel
      private(s0,ss,gosa1) etc.
   iteration: DO loop = 1, nn
!$omp barrier
!$omp master
      gosa = 0d0
!$omp end master
      gosa1 = 0d0
!$omp do

      <stencil loopnest>
!$omp end do

!$omp do
      <copy loopnest>
!$omp end do nowait

!$omp critical
      gosa = gosa + gosa1
!$omp end critical
   ENDDO iteration
!$omp end parallel
```

Aside: OpenACC versus OpenMP

- **One OpenMP threadteam for whole iteration loop**
  - OpenMP threads are "heavyweight" (expensive to create teams)
  - need barrier/master/critical regions for synchronisation
  - gosa1 is a thread-private variable; critical region used for reduction

- **Multiple OpenACC parallel loop kernels**
  - GPU kernels execute as threadblocks on different SMs
    - need separate kernels to synchronise
      - no global synchronisation method within a kernel
  - GPU threads are "lightweight" (kernels cheap to launch)
  - gosa1 is a shared reduction loop variable

- **Can we use a single OpenACC parallel region?**
  - Containing scalar code and multiple loop nests?
  - Analogous to version of code with multiple OpenMP parallel region
  - **NO!** Race conditions between kernels and with scalar code
    - think of acc end loop as comparable to omp end do nowait

Results using OpenMP across the CPU

language	Fortran		C	
precision	single	double	single	double
v00	2881	1454	2287	1131
v01	1177	565	1178	594
v02	37525	20300	37143	20287
v03	51921	28863	51078	28891
OMP16	8996	5416	9761	5086

- **Fastest OpenMP version uses 16 threads**
  - but only 3-4x faster than serial version
  - (code is quite memory bandwidth bound)
- **Overall speedup: 5-6x**
  - This is the sort of figure we expect
    - Kepler GPU memory bandwidth around 6x compared to Interlagos CPU

In summary

- **We ported the entire Himeno code to the GPU**
  - chiefly to avoid data transfers
    - 4 OpenACC kernels (only 1 significant for compute performance)
    - 1 outer data region
    - 2 inner data regions (nested within this)
  - 7 directive pairs for 200 lines of Fortran
  - Profiling frequently showed the bottlenecks
  - Correctness was also frequently checked
- **Data transfers were optimised at the first step**
- **We checked the kernels were scheduling sensibly**

In summary... continued

- **Further performance tuning**
  - data region gave a 44x speedup; kernel tuning is secondary
  - Low-level languages like CUDA
    - offer more direct control of the hardware
    - but OpenACC is much easier to use
      - should get close to CUDA performance
  - Remember Amdahl's Law:
    - speed up the compute of a parallel application,
      - and soon become network bound
    - Don't waste time trying to get an extra 10% in the compute
    - You are better concentrating your efforts on tuning the comms or I/O

- **Bottom line:**
  - 5-6x speedup from 7 directive pairs in 200 lines of Fortran/C
  - compared to the complete CPU

Now it's your turn

- **In the next practical, you will go through the steps of accelerating this same code**

- **You should:**
  - Check that you understand how to OpenACC a simple code
    - Either edit v00 to add the directives yourself
    - Or read and compile prepared v01, v02, v03 files and see it makes sense
    - Verify the results that I quote here for CCE

OpenACC Performance Tuning

Contents

- **Here we talk about a few tuning tips for OpenACC**
  - And introduce some more advanced OpenACC concepts

- **Outline of the lecture**
  - The **Golden Rules** of Tuning
    - where to get information to help you tune OpenACC
  - Tuning data locality
    - do this first!
  - Tuning kernels
    - correcting obvious scheduling errors
    - advanced schedule tuning (collapse, worker, vector_length clauses)
    - caching data (cache, tile directives)
    - Case Study: tuning scalar Himeno code
  - Tuning applications
    - asynchronicity and task dependency trees (async, wait clauses)
  - More extreme tuning
    - source code changes, reordering data structures, using CUDA

Before you go any further...

- **Don't start using the concepts in this lecture until:**
  - You have a correct OpenACC port with data movements minimised
    - i.e. you are bronze-level certified
- **Performance tuning may introduce bugs**
  - So you need a correct code from which to start
- **Performance tuning benefits are minimal...**
  - compared to the slow-down from unnecessary data-sloshing

Tuning code performance

- **Remember the Golden Rules of performance tuning:**
  - always profile the code yourself
    - always verify claims like "this is always the slow routine";
    - codes/computers change
  - optimise the real problem running on the production system
    - a small testcase running on a laptop will have a very different profile
  - optimise the right parts of the code
    - the bits that take the most time
    - even if these are not the exciting bits of the code
    - e.g. it might not be GPU compute; it might be comms (MPI), I/O...
  - keep on profiling
    - the balance of CPU/GPU/comms/IO will change as you go
    - refocus your efforts appropriately
  - Keep on checking for correctness
  - Know when to stop (and when to start again)

Tuning OpenACC performance

- **Tuning needs input:**
  - There are three main sources of information
    - make sure you use them
  - Compiler feedback (static analysis)
    - -hlist=a for CCE
    - -Minfo=accel for PGI
  - Runtime commentary
    - CCE only: CRAY_ACC_DEBUG=1 or 2 or 3
  - Code profiling, using tools like
    - CrayPAT and Cray Reveal
    - Nvidia compute profiler
    - pgprof for PGI

Tuning OpenACC codes

- **The main optimisation is minimising data movements**
- **How can I tell if data locality is important?**
  - CrayPAT will show the proportion of time spent in data transfers
    - May need to compile CCE with -hacc_model=auto_async_none to see this
  - Loopmark comments will tell you which arrays might be transferred
    - Compile CCE with -hlist=a and look at .lst files
  - Runtime commentary will tell you which arrays actually moved
    - and how often and when in the code
    - Compile as usual, export/setenv CRAY_ACC_DEBUG=2 at runtime
      - use the runtime API to control the amount of information produced

Tuning OpenACC data locality

- **What can I do?**
  - Use data regions to keep data resident on the accelerator
    - Understanding how data flows in application call tree is crucial, but tricky
  - Only transfer the data you need
    - if only need to transfer some of an array (e.g. halo data, debugging values),
    - rather than use copy* clause, use create and explicit update directives
    - packing/sending a buffer may be faster than sending strided array section

Tuning OpenACC data locality

- **What else can I do?**
  - Overlap data transfers with other, independent activities
    - use async clause on update directive; then wait for completion later
    - typical situations:
      - pipelining; send one chunk while another processes on the GPU
      - task-based overlap; can be hard to arrange
        - typical use case: pack halo buffer and transfer to CPU while GPU updates bulk
    - We'll discuss this in more detail later in this lecture
      - and give a practical example later in the tutorial
  - Beware of GPU memory allocation overheads
    - even create clause can have a big overhead for frequent, large arrays
    - maybe keep array(s) allocated between calls (add to higher data region)
      - add it to a higher data region as create and use present clause in subprogram
    - (not good for a memory-bound code, of course)

The next step

- **Once data movements have been optimised**
  - the next step is to improve the kernel scheduling
    - understand how the iterations of the loopnest are divided between threads
      - This is called "partitioning" of the loops
    - and how the threads are executed on the hardware
  - then we can improve this scheduling
- **For this, we need to understand how the hardware works**
  - we'll concentrate on Nvidia GPUs for this

How a kernel is executed on a GPU

- **A GPU kernel is executed by a large pool of threads**
  - Whether we use OpenACC or CUDA
  - The threads are logically divided into sets called threadblocks
  - Each threadblock executes on a different piece of hardware
    - Symmetric Multiprocessor (SM)
    - (Almost) like a single vector processor

- **Within a threadblock**
  - The threads are logically divided into sets called warps
    - threads within a warp (32 threads) are executed concurrently
    - (almost) like a set of vector instructions, each of fixed width 32
  - so the threadblock executes as a sequence of concurrent warps

- **The CUDA programming model**
  - does not make warps explicit
  - but you need to know about them to understand code performance

OpenACC scheduling

- **OpenACC makes the distinction explicit**
  - gang refers to threadblocks (just as in CUDA)
  - worker refers to entire warps
    - i.e. entire width-32 vector instructions (32 is fixed by the hardware)
  - vector refers to threads within a warp
    - i.e. 32 threads that do the same thing at the same time
- **When a loopnest is partitioned by the compiler**
  - The loop iterations are divided up into gangs of workers
    - each worker executes sets of vector instructions
  - There are many different ways for the compiler to do this
    - If you give no further instruction, the compiler will make a decision
    - You can see what this was from the compiler feedback
    - You can then over-ride this decision (at the next compilation)
      - by adding additional OpenACC directives and clauses
- **This is where OpenACC differs from OpenMP**
  - OpenMP parallel regions only partition one loop over the CPU threads
  - OpenACC parallel/kernels regions partition up to three nested loops

Kernel optimisation

- **So, after data movement, our next optimisation:**
  - make sure all the kernels vectorise
- **How can I tell if there is a problem?**
  - if a kernel is surprisingly slow on accelerator
    - in a wildly different place in the profile compared to running on CPU
  - then examine the loopmark compiler commentary files
- **What should I see in the loopmark**
  - loop iterations should be divided over
    - both the threads in a threadblock (vector)
    - and the threadblocks (gang)
- **With CCE, you should see either:**
  - For a single loop:
    - should be divided over both levels of parallelism
    - look for: Gg
  - For a loopnest:
    - 2 (or maybe 3) loops should be divided:
    - look for G and 2 or 3 g-s (maybe with numbers between)

Kernel optimisation

- **Making sure all the kernels vectorise**
  - should have one loop "partitioned across threads within a threadblock"
- **generally want to vectorise the innermost loop**
  - usually fastest-moving array index, for coalescing
- **if it is not vectorised, can we make it vectorise?**
  - Can loop iterations be computed in any order?
    - if not, rewrite code
      - avoid loop-carried dependencies
      - e.g. buffer packing: calculate rather than increment
      - these rewrites will probably perform better on CPU
    - there is a lot of literature on vectorised algorithms

Replace:

```
i = 0
DO y = 2,N-1
   i = i+1
   buffer(i) = a(2,y)
ENDDO
buffsize = i
```

By:

```
DO y = 2,N-1
   buffer(y-1) = a(2,y)
ENDDO
buffsize = N-2
```

Forcing compiler to vectorise

- **If the loop is already vectorisable, guide the compiler**
  - Either a gentle hint:
    - put "acc loop independent" directive above this loop
      - recompile the routine and check compiler feedback to see if this worked
  - Or a direct order:
    - put "acc loop vector" directive above this loop
  - check the code is still correct (as well as running faster)
    - the compiler might not be vectorising the loop for a good reason
- **Note:**
  - you can put extra loop directives inside a parallel or kernels region
    - usually do this to change scheduling by adding tuning clauses
    - remember reduction clauses
      - you will need to duplicate any reduction clauses on these new directives

Changing the vectorisation with seq

- **If the inner loop is vectorising but performance is still bad**
  - is the inner loop really the one to vectorise in this case?
    - in this example, we should vectorise the i-loop
      - because we happen to know mmax is small here
        - or because CrayPAT loop-level profiling told us
    - put "acc loop seq" directive above m-loop
      - then executed "redundantly" by every thread
      - also t is now an i-loop private scalar
        - rather than a reduction variable
        - this should also help performance

```
!$acc parallel loop
DO i = 1,N
   t = 0
!$acc loop seq
   DO m = 1,mmax
      t = t + c(m,i)
   ENDDO
   a(i) = t
ENDDO
!$acc end parallel loop
```

- **The compiler should now vectorise a different loop**
  - usually the next one up in the loopnest
    - in this case there is no choice but the i-loop
  - if not, you will need an "acc loop vector" directive in the right place

An aside on this example

```
!$acc parallel loop
DO i = 1,N
   t = 0
!$acc loop seq
   DO m = 1,mmax
      t = t + c(m,i)
   ENDDO
   a(i) = t
ENDDO
!$acc end parallel loop
```

- **We forced vectorisation of the i-loop**
  - because mmax was too small
    - small loops will not give the GPU enough work
- **Performance is still likely to be bad**
  - a warp is 32 threads, executing as a vector
    - each thread has a different i-value
  - data for the warp is done in vector loads
    - but c(m,1:32) is not a contiguous chunk of memory
    - so we will need multiple vector loads for each warp
  - loads/stores from global memory are slow, so performance will suffer
    - this is a hardware feature, not a limitation of OpenACC
- **So what can we do?**
  - Either re-arrange the affected data arrays to get coalescing
    - so warps get their data in the minimum vector loads/stores
    - this means we want the vector index to be fastest-moving array index
      - c(i,m) does give coalesced memory accesses;
      - because c(1:32,m) is a contiguous chunk of memory
    - But this means refactoring your code (which may or may not be much work)
      - You may get better CPU performance as well (e.g. from AVX instructions)
  - Or you could try using the cache clause for array c

It's all vectorising, but still performing badly

- **Profile the code and start "whacking moles"**
  - optimise the thing that is taking the time
  - if it really is a GPU compute kernels...
- **GPUs need lots of parallel tasks to work well**
- **First look at loop scheduling using OpenACC clauses**
- **Then might need to consider more extreme measures**
  - source code changes
  - handcoding CUDA kernels

Over-riding the default scheduling

- **You can change how loops are scheduled**
  - by using additional loop directives above the relevant loops
  - with clauses to tell the compiler how to schedule that loop
    - use clauses: gang, worker, vector
    - we've already seen vector in use in the previous example
    - you can also use the seq clause
      - tells compiler what not to do, allowing it freedom in scheduling remaining loops
  - Can put more than one of gang, worker, vector clauses on single loop
    - That loop's iterations are then divided over multiple levels of parallelism
    - To schedule one loop over the entire GPU:
      - acc loop gang worker vector
  - Can only mention gang, worker, vector (at most) once each per kernel
    - any not mentioned will be handled automatically by the compiler
      - it will choose where best to apply them
    - once all levels of parallelism are used up
      - additional loops will execute sequentially (redundantly)

Changing the partitioning

- **Before we alter the schedule, two easy tuning choices**
  - changing the number of threads per threadblock
  - changing the total number of threadblocks used

- **Do this with clauses on the parallel/kernels directives**
  - vector_length changes the number of threads per threadblock
    - CCE allowed values are: 1, 32, 64, 128, 256, 512, 1024
    - CCE default value is 128
  - num_gangs changes the number of threadblocks
    - You do not have to specify this (unlike CUDA)
    - The compiler will make a default choice of sufficient blocks

- **Handy tip:**
  - To debug a kernel by running on a single GPU thread, use:
    - !$acc parallel loop gang worker vector num_gangs(1) vector_length(1)
  - Useful for checking race conditions in parallelised loopnests
    - but execution will be very slow, so maybe find a cut-down testcase first

vector_length bigger than 32?

- **What happens if vector_length is larger than 32?**
  - warps execute vector instructions of width 32 (fixed by the hardware)
  - if vector_length is more than 32
    - multiple vector instructions are used to implement the operations
    - i.e. the vector is decomposed into multiple warps
      - or multiple workers, in the language of OpenACC
  - this is exactly what happens implicitly in CUDA

Advanced loop scheduling

- **OpenACC loop schedules are limited by the loop bounds**
  - one loop's iterations are divided over threadblocks
  - another loop's iterations are divided over threads in threadblock

- **This has limitations, for instance**
  - "tall, skinny" loopnests (j=1:big; i=1:small) won't schedule well
    - inner loop is too small
      - if less than 32 iterations won't even fill a warp, so wasted SIMT
  - "short, fat" loopnests (j=1:small; i=1:big) also not good
    - outer loop is too small
      - want lots of threadblocks to swap amongst SMs
  - In both cases there are enough total iterations to keep the GPU busy
    - but division of iterations between the two loops is non-optimal for the GPU

- **How do we better spread the loop iterations over the GPU?**
  - collapse clause
  - worker clause

collapse clause

- **A way of increasing OpenACC scheduling flexibility**
- **Merges iterations of two or more loops together**
  - We then apply OpenACC scheduling clauses to the composite loop
  - Both "tall, skinny" and "short, fat" examples will benefit from
    - acc loop collapse(2) gang worker vector
      - collapse the j and i loops into a single iteration space
        - DO ij=1,small*big
      - then divide the total iterations over all levels of parallelism on the GPU
  - Things to look for
    - the compiler may use this automatically (look for C in loopmark feedback)
    - no guarantee that it is faster
      - index rediscovery (if needed) requires expensive integer divisions
        - e.g. j = INT(ij/big); i = ij - j*big
    - you can only collapse perfectly nested loops
  - The next slides give some examples of using the clause

Examples of using the collapse clause

- **Consider a three-level loopnest (i inside j inside k)**
  - loopnest should be perfectly nested to use collapse clause

- **Here are a few examples of what you might do:**
  - Collapse all three loops and schedule across GPU
    - "acc parallel loop collapse(3) gang worker vector" above k-loop
      - You don't really need "gang worker vector" here
        - there's only one composite loop left to partition after collapsing
    - Often the best way to improve performance of small tripcount loopnests
      - removes restriction of levels of parallelism tied to specific loops

  - Schedule inner two loops over threads in threadblock
    - "acc parallel loop gang" above k-loop
    - "acc loop collapse(2) vector" above j-loop
      - don't need "gang"; enough warps are used to cover all the iterations
    - Do this if the k-loop is quite large, but the i-loop is small
      - but you often see that collapse(3) is better

More examples using the collapse clause

- **Here are a few more examples of what you might do:**
  - Keep with the three-level loopnest (i inside j inside k)
  - In most cases these do not give the best performance

  - Schedule outer two loops over the threadblocks
    - "acc parallel loop collapse(2) gang" above k-loop
    - optionally also put "acc loop vector" above i-loop

  - Schedule outer two loops together over entire GPU
    - "acc loop collapse(2) gang worker vector" above k-loop
    - optionally also put "acc loop seq" above i-loop

  - Schedule k-loop and i-loop together over entire GPU
    - i.e. you want to collapse just the i- and k-loops
    - you can't:
      - collapsed loops must be perfectly nested (i.e. consecutive statements in the code)
      - so you'll need to reorder the loops in the code first

Explicit worker clauses

- **We've already seen the worker clause**
  - as part of "gang worker vector", meaning "everything"
- **Often, the compiler partitions two loops**
  - the outermost loop over gangs
  - the innermost loop over vector
  - remaining loops are executed redundantly (sequentially) by all threads
  - if the innermost loop has more than 32 iterations
    - it is automatically split into multiple workers
- **We can use the worker clause to partition a third loop**
  - Doesn't change the total work (number of loopnest iterations)
  - But does change how work distributed between threads
  - The next slide explain this in detail, which you may wish to skip

Scheduling with and without the worker clause

- **The default scheduling**
  - k-loop iterations divided over threadblocks
  - i-loop iterations divided within a threadblock
    - round-robin distribution
      - first thread does i=1, V+1, 2*V+1, ...
      - V is vector_length value (default 128 with CCE)
    - threads automatically grouped into warps
      - first warp does i=1:32, V+1:V+32, ...
  - each thread does all the j-loop iterations

```
!$acc parallel
!$acc loop gang
DO k = 1,N
!$acc loop seq !implicit
   DO j = 1,N
!$acc loop vector
      DO i = 1,N
```

- **With explicit loop worker directive**
  - k-loop divided as before
  - i-loop iterations are divided within a warp
    - first thread in warp does i=1, 33, 65, ...
    - each warp does *all* values: i=1:32, 33:64, ...
  - j-loop iterations divided over warps
    - number of warps, W (see previous):
      - either: num_workers value
      - or: vector_length value divided by 32
    - round-robin distribution
      - first warp does j=1, W+1, 2*W+1, ...

```
!$acc parallel
!$acc loop gang
DO k = 1,N
!$acc loop worker
   DO j = 1,N
!$acc loop vector
      DO i = 1,N
```

num_workers clause

- parallel/kernels can also take a num_workers clause
  - specifying vector_length(NTHREADS>32)
  - is equivalent to specifying num_workers(NTHREADS/32)
    - plus vector_length(32)
  - You probably only want to specify explicitly num_workers
    - if you also put in an "acc loop worker" directive
    - in this case, only vector_length(32) makes sense

collapse or worker?

- **Both try to increase the scheduling flexibility**
- **Perfectly nested loops with one or more low tripcounts**
  - probably better to use the collapse clause
- **Imperfectly nested loops with one or more low tripcounts**
  - may benefit to put "!$acc loop worker" on the middle loop
    - collapse won't work here
- **But it is difficult to predict which will be best**
  - You may need to try both
  - We'll give an example in this lecture

The cache directive

- **Main accelerator memory is relatively slow**
  - If a data element is accessed multiple times
  - It may make sense to temporarily store it in a faster cache
  - typical use case is a finite difference stencil, e.g.
    - for all `i`, calculate: `a[i]=b[i+1]-2*b[i]+b[i-1]`
      - `b[3]` is used three times, to calculate all of: `a[2]`, `a[3]`, `a[4]`.
    - so we would like to cache values of `b` for reuse by other loop iterations

- **The cache directive suggests this to the compiler**
  - suggests, but does not compel: compiler can ignore suggestion
    - e.g. if you try to store more data than the cache can hold
    - check compiler feedback to see what it did
  - No guarantee that this improves performance
    - you may be polluting some automatic caching (hardware or software)

- **Nvidia GPUs**
  - threads within a threadblock execute on single SM piece of hardware
  - have joint access to a common, fast, but small, "shared memory"

cache clause example

- **A first example:**
  - loop-based stencil
    - bigger than before
    - size: 2*RADIUS+1
  - inner loop must be sequential
  - RADIUS should be known at compile time (parameter or cpp)

```
!$acc parallel loop copyin(c)
  DO i = 1,N
   result = 0
!$acc cache(in(i-RADIUS:i+RADIUS),c)
!$acc loop seq
   DO j = -RADIUS,RADIUS
    result = result + c(j)*in(i+j)
   ENDDO
   out(i) = result
  ENDDO
```

Multi-dimensional loopnests

- **How much data should be cached here?**
  - B(NI-1:NI+1,j) would probably fill the cache for one j value
  - And we wouldn't get any re-use for j±1
- **To use the cache clause here**
  - Need to tile the loopnest
  - The compiler may do this
  - Or we may want more explicit control

```
!$acc parallel loop
DO k = 1,NK
 DO j = 1,NJ
  DO i = 1,NI
!$acc cache( B(i-1:i+1,j-1:j+1,k) )

      A(i,j,k) = B(i,  j,  k) - &
                 ( B(i-1,j-1,k) &
                 + B(i-1,j+1,k) &
                 + B(i+1,j-1,k) &
                 + B(i+1,j+1,k) ) / 5
    ENDDO
  ENDDO
ENDDO
!$acc end parallel
```

The tile clause

- **Added to loop directive**
  - New in OpenACC v2.0
- **Blocks loops in a loopnest**
  - Using specified blocking factors
    - fixed at compile-time
    - Or * to use compiler default
    - one argument per loop
  - Outer tile loops
    - migrated to outside of nest
  - Inner element loops
    - have known size

```
!$acc loop tile(8,16)   ! next 2 loops
DO j = 1,NJ
 DO i = 1,NI
```

```
! equivalent explicit code
!$acc loop
DO jtile = 1,NJ,16
 DO itile = 1,NI,8

  DO j = jtile,jtile+16-1
   DO i = itile,itile+8-1
```

- **Scheduling clauses on loop directive still apply, if used**
  - gang applies to outer, tile loops
  - vector applies to inner, element loops
    - probably want composite tilesize to be multiple of 32
  - worker might apply to either, depending on how used
    - see standard for details

Multi-dimensional caching

- **The corrected version:**
  - j-loop: tilesize=16
  - i-loop tilesize=64
  - automatic scheduling:
    - outer tile loops: worker
    - inner element loops: vector
  - cached data:
    - (64+2)*(16+2)=1188 elements
      - 4kB of floats or integers
      - 8kB of doubles

```
!$acc parallel loop gang
DO k = 1,NK
!$acc loop tile(64,16) worker vector
  DO j = 1,NJ
    DO i = 1,NI
!$acc cache( B(i-1:i+1,j-1:j+1,k) )

      A(i,j,k) = B(i,  j,  k) - &
                 ( B(i-1,j-1,k) &
                 + B(i-1,j+1,k) &
                 + B(i+1,j-1,k) &
                 + B(i+1,j+1,k) ) / 5
    ENDDO
  ENDDO
ENDDO
!$acc end parallel
```

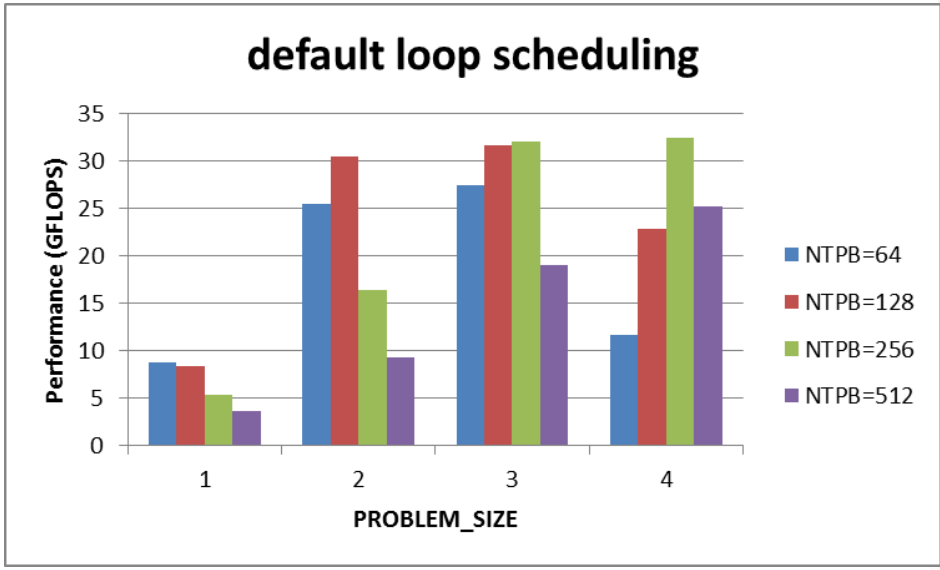
Tuning performance with the loop schedule

- **Tuning the loop schedule is main performance tuning**
  - first make sure you've minimised data transfers
  - then make sure the default schedule is sensible
    - loop(s) partitioned both both ACROSS and WITHIN threadblocks
  - then think about using tuning clauses to improve performance
- **The next few slides give a tuning Case Study**
  - The scalar Himeno code that we discussed earlier
  - Applying OpenACC tuning clauses to the jacobi stencil kernel

Scalar Himeno performance

- **PROBLEM_SIZE=1,2,3,4**
  - For a parallel problem, this simulates strong scaling choices
  - Here we look at all 4 options (all double precision)
- **vector_length choices: NTPB=64,128,256,512**
  - Easy tuning choice, but needs a recompile with CCE
    - you need to put in vector_length(NTPB) clauses on kernels
    - and compile with -DNTPB=<value>
  - CCE defaults to 128

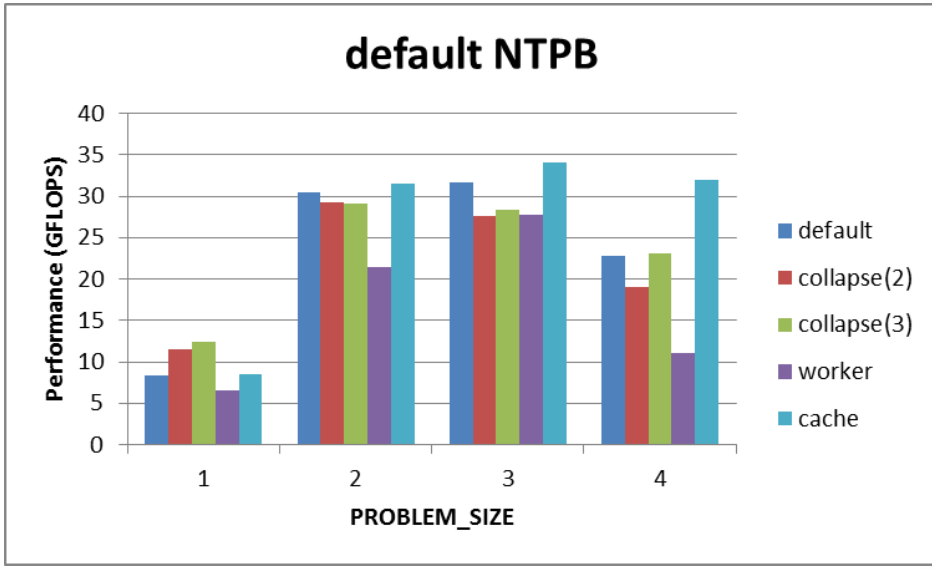
- **This has a BIG effect**
  - best NTPB relative to default:
  - PROBLEM_SIZE=1: + 5%
  - PROBLEM_SIZE=2: default
  - PROBLEM_SIZE=3: + 2%
  - PROBLEM_SIZE=4: +42%



Scalar Himeno performance

- **PROBLEM_SIZE=1,2,3,4**
- **5 algorithm choices for jacobi stencil kernel**
  - default scheduling
  - collapse(2) inner i,j-loops
  - collapse(3) all loops
  - worker schedule for j-loop
  - cache(p(i-1:i+1,j,k)) with default loop scheduling
- **vector_length: default**

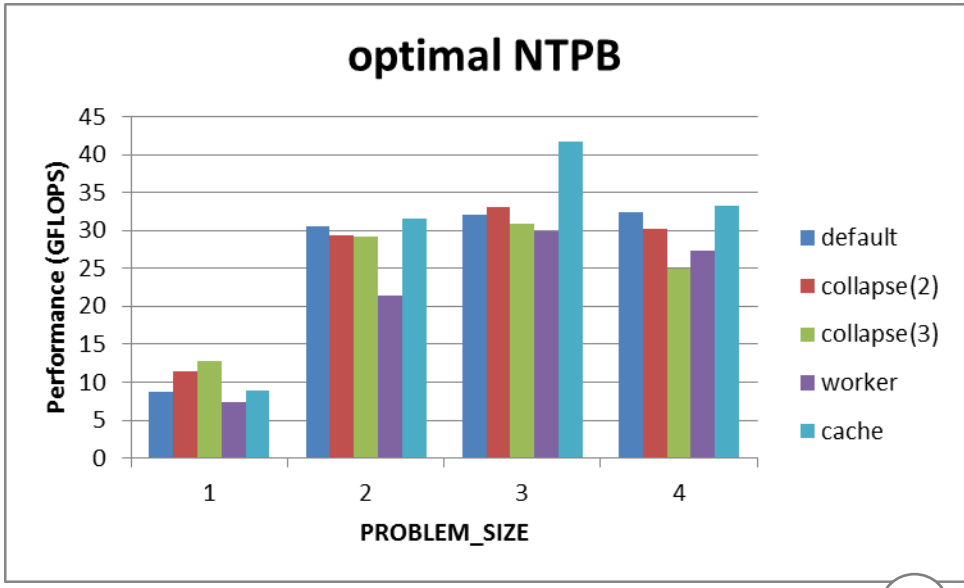
- **Effect also quite big**
  - best NTPB relative to default:
  - PROBLEM_SIZE=1: +48%
  - PROBLEM_SIZE=2: + 4%
  - PROBLEM_SIZE=3: + 8%
  - PROBLEM_SIZE=4: +41%



Scalar Himeno performance

- PROBLEM_SIZE=1,2,3,4
- 4 algorithm choices for jacobi stencil kernel
  - default scheduling
  - collapse(2) inner i,j-loops
  - collapse(3) all loops
  - worker schedule for j-loop
  - cache(p(i-1:i+1,j,k)) with default loop scheduling
- vector_length: now use best for each algorithm choice

- Final improvements
  - compared to default
    - scheduling and vector_length:
  - PROBLEM_SIZE=1: +52%
  - PROBLEM_SIZE=2: + 4%
  - PROBLEM_SIZE=3: +32%
  - PROBLEM_SIZE=4: +46%



Scalar Himeno tuning conclusions

- **Tuning can have a big effect, relative to default**
  - It is worth doing, but only after you optimise the data locality
    - data region gave 44x speedup; kernel tuning gave less than 2x
  - We gained something at every problem size (don't reject a mere 2x!)
  - Only explored vector_length and basic scheduling (collapse, worker)
    - only change was OpenACC directives; CPU version same

- **General conclusions for scalar Himeno:**
  - Larger vector_length suited larger problem sizes
    - 64 best for PS1; 128 best for PS2; 256 best for PS3,4
  - Once we had optimised vector_length
    - collapse(3) was best for small problems (PS1)
    - cache was best for large problems (PS2-4)
      - caching in j,k directions did not improve performance

- **What else could we try?**
  - async clause? no; scalar Himeno is too simple for task-based overlap
    - but we will see that async is very important in the parallel case

Exploiting overlap

- **If individual kernels are performing well**
  - Can you start overlapping different computational tasks?
    - kernels on the GPU
    - data transfers to/from the GPU
    - maybe even separate computation on the CPU
- **You can do this using the async features of OpenACC**
  - very similar to streams in CUDA

Asynchronicity

- **GPU operations are launched asynchronously from CPU**
  - control returns immediately to the host
  - host must then wait or test for completion
    - automatically (e.g. handled by OpenACC compiler)
    - manually (e.g. via OpenACC directives or API calls)
  - applies to:
    - computational kernels: parallel and kernels regions
    - PCIe data transfers: update directives
    - plus some other directives

- **Synchronisation**
  - By default, the compiler will handle synchronisation
    - may wait for every operation to complete, or
    - may be smarter, and use minimum number of waits needed for correctness
    - CCE: control with compiler option -hacc_model=auto_async_*

User control via async clause

- **The async clause overrides the default behaviour**
  - User is now in control of synchronisation
    - applies to: parallel, kernels, update directives
  - User needs to insert appropriate synchronisation points
    - via wait directive or OpenACC API calls
    - N.B. there is no implicit wait at the end of a subprogram!

- **Between synchronisation points**
  - May get overlap of concurrently-executing kernels
    - depending on hardware
  - Should get overlap of GPU computation with data transfers
    - in both directions on PCIe bus
  - Data transfers in the same direction will not overlap
    - runtime will serialise the transfers

  - async operations queued in advance should improve throughput
    - even if the operations themselves are serialising

A stream of tasks

- **The async clause can take a handle:**
  - async(handle) where handle is a (positive or zero) integer
- **Operations launched with the same handle**
  - known as a stream of tasks
  - guaranteed to execute sequentially in the order they were launched
  - not guaranteed to execute immediately after each other
    - there could be delays
- **Synchronisation**
  - wait(handle) directive ensures that stream of tasks has completed
  - can use an API call to do same thing
    - another API call can be used to test whether stream has completed
- **N.B.**
  - If you've used CUDA streams, these concepts should be very familiar

Multiple streams of tasks

- **You can launch multiple streams of tasks at once**
  - each with a different handle value
  - Tasks within a given stream
    - guaranteed to execute sequentially
  - Tasks in different streams
    - may overlap or serialise, as the hardware and runtime allows
    - operations in different streams should be independent
      - or we have a race condition

- **Synchronisation**
  - wait(handle) directive ensures one stream has completed
    - wait(handle1,handle2,...) can be used to finalise multiple streams
    - wait directive with no argument ensures all streams have completed
  - API calls can be used to wait on, or test for, all these completion cases

- **Nvidia GPUs (Kepler, Fermi)**
  - currently support up to 16 simultaneous streams in hardware
  - if handle is too large, runtime MODs it back into allowable range
    - so handle=16 same as handle=0
    - this can lead to false dependencies between streams

OpenACC async clause

- **async[(handle)] clause**
  - for parallel, kernels, 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 Kepler (and Fermi)
  - streams identified by handle (integer-valued)
    - tasks with same handle execute sequentially
    - can wait on one, more or all tasks

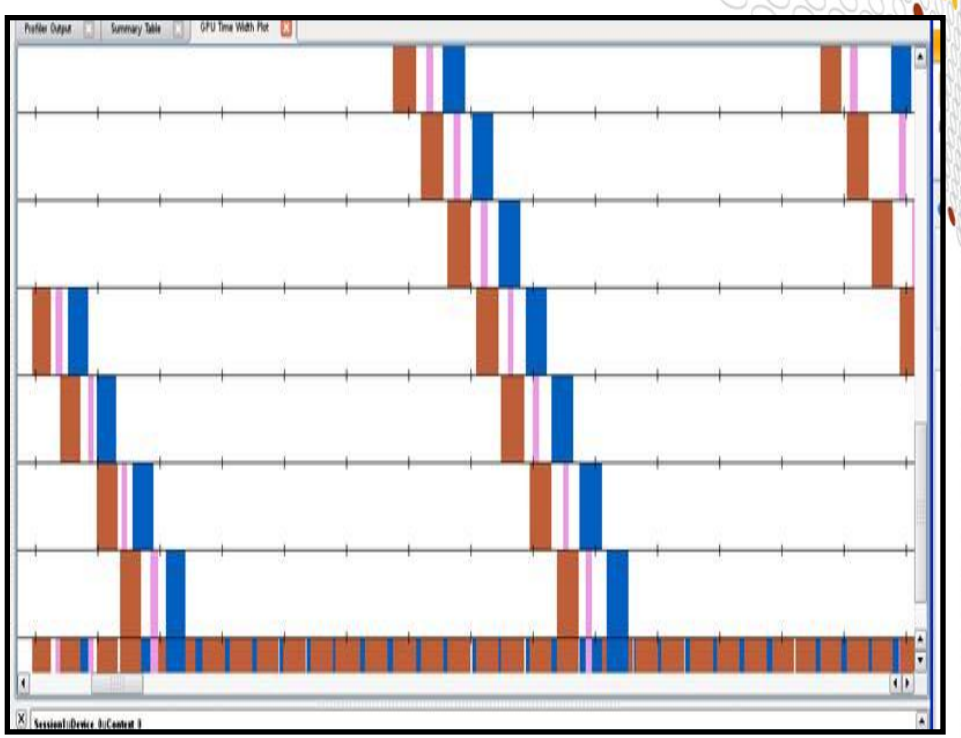
- **!$acc wait: waits for completion of all streams of tasks**
  - !$acc wait(handle) waits for a specified stream to complete
- **Runtime API library functions**
  - can also be used to wait or test for completion

OpenACC async first example

- **First example**
  - 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)

```
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:**
  - CPU: 3.76s
  - OpenACC, blocking: 1.10s
  - OpenACC, async: 0.34s

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

- **Alternative to pipelining is task-based overlap**
  - Harder to arrange; needs knowledge of data flow in specific application
  - May (probably will) require application restructuring (maybe helps CPU)
  - Some results later in Himeno Case Study

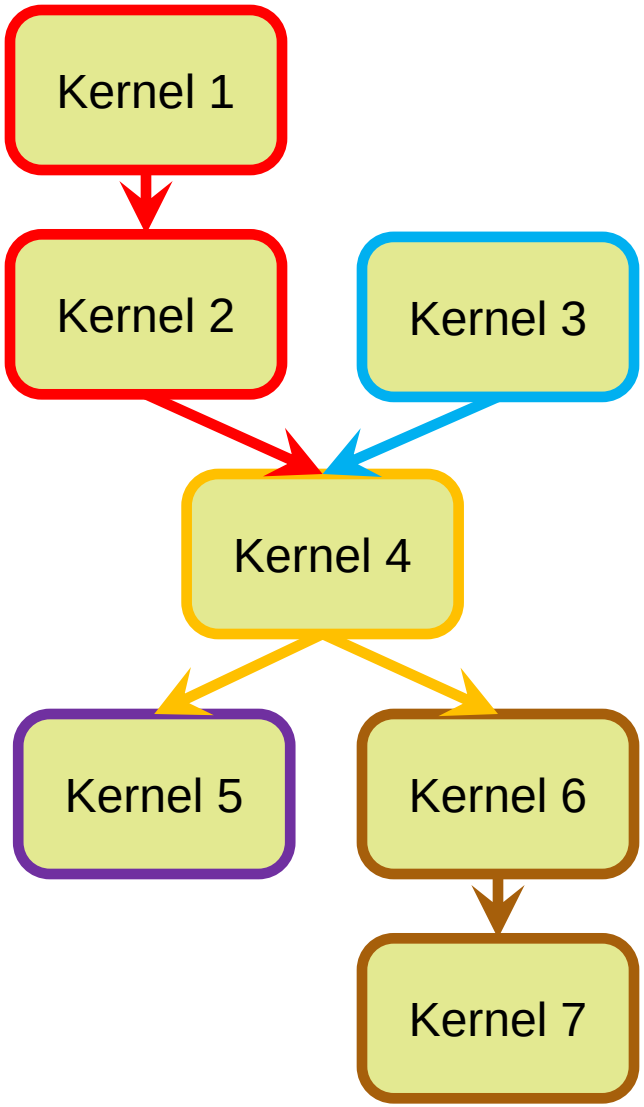
Going further with OpenACC v2.0

- **OpenACC v1.0:**
  - async/wait perfect for handling linear streams of dependent tasks
  - but no way to set up more complicated dependency tree
    - e.g. to say:
      - "When you have finished these streams of tasks, start this one"
      - "When you have finished this stream of tasks, start these ones"
  - dependencies like this have to be handled by the host
    - extra host-side synchronisation points:
      - reduce the performance of the code
      - reduce developer's ability to use CPU for other tasks in the code

- **OpenACC v2.0:**
  - now allows you to set up dependency tree
  - wait(handle) clause for parallel, kernels, update directives
  - async(handle) clause for wait directive

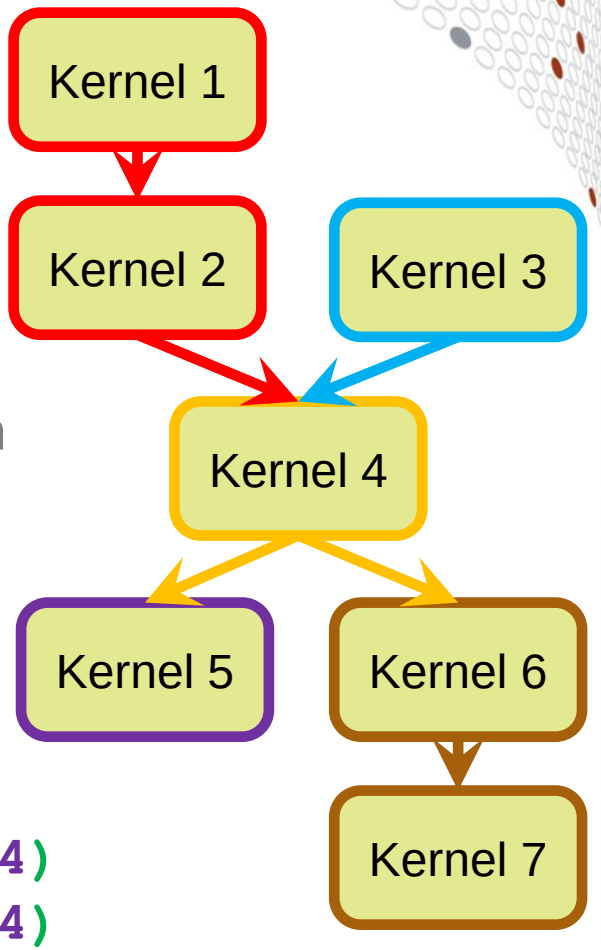
High-level example

```
!$acc parallel loop async(stream1)
<Kernel 1>
!$acc parallel loop async(stream1)
<Kernel 2>

!$acc parallel loop async(stream2)
<Kernel 3>

!$acc parallel loop async(stream3) &
!$acc     wait(stream1,stream2)
<Kernel 4>

!$acc parallel loop async(stream4) &
!$acc     wait(stream3)
<Kernel 5>

!$acc parallel loop async(stream5) &
!$acc     wait(stream3)
<Kernel 6>

!$acc parallel loop async(stream5)
<Kernel 7>

!$acc wait ! ensures all completed
```



Advanced asynchronous concepts

- **Could reuse streams**
  - e.g.
    - `stream4=stream1`
    - `stream5=stream2`
  - dependencies are determined by order in which the tasks are launched
  - but this is likely to make your head hurt
    - especially if you have a multi-threaded program



- **Asynchronous waits ?!**
  - This:
    - `!$acc wait(stream3) async(stream4)`
    - `!$acc parallel loop async(stream4)`
  - is equivalent to:
    - `!$acc parallel loop async(stream4) wait(stream3)`

Using dependency trees

- **The compiler may use some async automatically**
  - See our earlier comment
- **But, if you want**
  - multiple, overlapping streams of tasks
  - and, potentially, a more complicated dependency tree
- **Then you will need to do it yourself**
  - This requires:
    - good knowledge of your code (so you know what it does, and where)
    - good knowledge of the algorithm (so you can change the code)
    - good knowledge of the science (so you can change the algorithm)
  - OpenACC improves productivity, but cannot replace the "hard thinking"
- **We'll show a real-world example later**
  - see the parallel Himeno code lecture

parallel loop vs. parallel and loop

- **parallel loop is a composite directive**
  - it is a short hand for consecutive parallel and loop directives
    - kernels loop is similarly a fusion of kernels and loop
- **It takes the same clauses as parallel and loop**
  - loop clauses will apply to the outermost loop in the loopnest
  - We can put extra loop directives inside for the other loops
- **We can write the two directives separately**
  - if we want; it's a matter of style
- **We can also go further if we want**
  - one parallel region can span multiple code blocks
  - Each code block could be a loopnest or serial code
- **Should we go further like this?**
  - Short answer: No (in my opinion)
  - The longer answer is on the next slide

The longer answer

- A parallel region starts with redundant execution
  - Until we do some partitioning, all the threads are doing the same thing
    - or maybe just by one thread per block (its implementation dependent)
  - loopnests in parallel region are not automatically partitioned
    - you need to explicitly use a loop directive for this to happen
      - it's easy to forget this
  - scalar code (serial code or loopnests without a loop directive)
    - executed redundantly, i.e. identically by every thread
  - There is no synchronisation between redundant code and/or kernels
    - this does offers potential for overlap of execution on GPU, but
    - also offers potential (and likelihood) of race conditions and incorrect code
  - There is no mechanism to put a barrier inside a parallel region
    - this is a hardware limitation, not OpenACC specific
    - the best you can do is to end the parallel region and start a new one

If you still think you want to

- **My advice: don't...**
  - GPU threads are very lightweight (unlike CPU threads with OpenMP)
    - so don't worry about having extra parallel loop regions
  - If you are trying to get overlap, use async, wait
    - does the same thing
    - with greater code clarity
    - and gives greater control

- **... but if you feel you really must**
  - begin with composite parallel loop and get correct code
    - separate the directives with care, and only as a later performance tuning
      - when you are sure the kernels are independent and with 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
  - and watch out for the common mistakes on the next slides

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
```

parallel loop vs. parallel and loop

- **My advice: when you actually might want to**
  - You *might* split the directive if:
    - you have a single loopnest, and
    - you need explicit control over the loop scheduling
    - you do this with multiple loop directives inside parallel region
      - or you could use parallel loop for the outermost loop, and loop for the others
- **But beware of reduction variables**
  - With separate loop directives, you need a reduction clause on every loop directive that includes a reduction:

```
t = 0
!$acc parallel loop &
!$acc    reduction(+:t)

DO j = 1,N

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

Correct!

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

Wrong!

```
t = 0
!$acc parallel

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

Wrong!

```
t = 0
!$acc parallel

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

Correct!

Extreme tuning

- **You've tried tuning with OpenACC clauses**
  - but you think a kernel's performance can still be improved further
  - (and this kernel is the real performance-limiter in your application)
- **Now (and only now) you may need extreme tuning**
- **Some examples:**
  - main source code changes
    - What changes will work? There is no definitive guide.
    - Following slides give two cases
  - mixing languages
    - You could handtune the slow kernel in CUDA
      - you'll need to work hard in CUDA to beat OpenACC, though
    - OpenACC allows full interoperability with CUDA (i.e. sharing data)
      - Can call CUDA from OpenACC: useful for hand-tuning individual kernels
      - Can call OpenACC from CUDA: useful if migrating some/all of code from CUDA
      - more details later in the course

Avoiding temporary arrays

- **Perfect loop nests often perform better than imperfect**
  - Imperfect loopnests often use temporary arrays
    - e.g. in a stencil like MultiGrid, to avoid additional duplicated computation
  - With OpenACC, these arrays are privatised; too big for shared memory
    - Imperfect loop nest also means scheduling decisions are restricted
- **Try two approaches; which (if any) faster depends on code**
  - Remove temporary arrays by manually inlining (eliminate array b)
    - one perfect loop nest; cache clause can use shared mem/regs where needed
  - Manually privatise arrays and split the loopnest (b(i)→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
```

More drastic performance optimisations

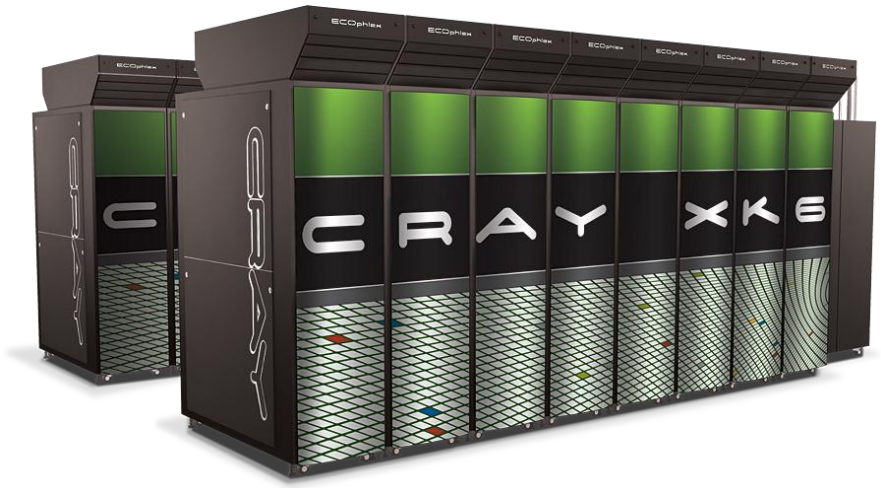
- **Would reordering your data structures help?**
- **For instance, consider a simulation of interacting particles**
  - Nmax particles, each having Smax internal properties
    - code separately combines the internal properties for each particle
  - CPU code usually stores data as f(Smax,Nmax) or f[Nmax][Smax]
    - good cache reuse when we access all the properties of a particle
  - GPU code would normally parallelise over the particles
    - each thread processes the internal properties of a single particle
    - first warp would attempt vector load of s^{th} prop. of first 32 particles: f(s,1:32)
    - no coalescing (vector load needs contiguous block of memory)
    - very poor performance (even if Smax is small)
  - Better to reorder data so site index fastest: fgpu(Nmax,Smax)
    - vector load of fgpu(1:32,s) now stride-1 in memory
    - if code memory-bandwidth-bound, you will see a big speed-up

- **Quite an effort to reorder data structures in the code**
  - but... may also see benefits on CPU
    - especially with AVX (and longer vectors in future CPU processors)

Conclusions

- **Tuning kernel performance can give benefits**
  - loop scheduling
  - caching data
  - asynchronicity and task dependency trees
- **But there are a lot of options to try**
  - hard to make general rules that apply in all cases
  - best options affected by local problem size
- **An autotuning approach may be best (offline or at runtime)**
  - e.g. as developed in the EU CRESTA exascale project
- **OpenACC is a big advantage here**
  - proto-typing in a low-level language is slow and painful
    - e.g. trying new loop schedule, caching in shared memory
  - In OpenACC you just change one or two directives

Performance Tools for OpenACC



Contents

- What information do you need when tuning?
- How can you get this information?

Analyzing an Application – Questions to Ask

- **What goes on within the time step loop?**
  - Where is computation
  - Where is communication
  - What data is used
- **What, if any computation can be moved?**
- **What, if any communication can be moved?**
- **Identification of potential overlap**
- **What about I/O**

What to Determine About Your Application

- **Where are the major arrays allocated and how are they accessed?**
  - WHY? – We need to understand how arrays can be allocated to assure most efficient access by major computational loops. (First touch, alignment, etc)
- **Where are the major computational and communication regions?**
  - WHY? – We want to maintain a balance between computation and communication. How much time is spent in a computational region, what if any communication can be performed during that time?
- **Where is the major I/O performed?**
  - WHY? – Can we perform I/O asynchronously with computation/communication?

Performance Feedback

- **Compiler feedback through listings (static analysis)**
  - CCE: -rm (Fortran) –hlist=a (C)
  - PGI: -Minfo=accel
- **Runtime commentary of accelerator activity**
  - CCE: setenv CRAY_ACC_DEBUG <1,2,3>
  - PGI: setenv PGI_ACC_TIME 1
- **Application profiling**
  - Cray Performance Toolsuite
  - Nvidia compute profiler
  - PGPROF® for PGI
  - Vampir Toolsuite

Compiler Feedback Through Listing

- `cc -rm file.f`

```
ftn-6413 ftn: ACCEL File = himeno_caf_acc.f08, Line = 292
  A data region was created at line 292 and ending at line 446.

ftn-6263 ftn: VECTOR File = himeno_caf_acc.f08, Line = 306
  A loop starting at line 306 was not vectorized because it contains a
reference to a non-vector intrinsic on line 396.

ftn-6405 ftn: ACCEL File = himeno_caf_acc.f08, Line = 310
  A region starting at line 310 and ending at line 338 was placed on the
accelerator.

ftn-6415 ftn: ACCEL File = himeno_caf_acc.f08, Line = 310
  Allocate memory and copy variable "wgosa" to accelerator, copy back at line
338 (acc_copy).

ftn-6430 ftn: ACCEL File = himeno_caf_acc.f08, Line = 311
  A loop starting at line 311 was partitioned across the thread blocks.

ftn-6509 ftn: ACCEL File = himeno_caf_acc.f08, Line = 312
  A loop starting at line 312 was not partitioned because a better candidate
was found at line 313.
```

Compiler Loopmark

- `cc -rm file.f`

```
71.    G------------< !$acc data present(a,b,c,p,wrk1,bnd)
72.    G G----------< !$acc parallel loop private(i,j,k)
73.    G G g--------<       do k=1,mkmax
74.  + G G g C------<          do j=1,mjmax
75.    G G g C gC---<             do i=1,mimax
76.    G G g C gC                    a(i,j,k,1)=0.0
77.    G G g C gC                    a(i,j,k,2)=0.0
78.    G G g C gC                    a(i,j,k,3)=0.0
79.    G G g C gC                    a(i,j,k,4)=0.0
80.    G G g C gC                    b(i,j,k,1)=0.0
81.    G G g C gC kk                   b(i,j,k,2)=0.0
82.    G G g C gC                    b(i,j,k,3)=0.0
83.    G G g C gC                    c(i,j,k,1)=0.0
84.    G G g C gC                    c(i,j,k,2)=0.0
85.    G G g C gC                    c(i,j,k,3)=0.0
86.    G G g C gC                    p(i,j,k) =0.0
87.    G G g C gC                    wrk1(i,j,k)=0.0
88.    G G g C gC                    bnd(i,j,k)=0.0
89.    G G g C gC--->             enddo
90.    G G g C------>          enddo
91.    G G g------>>       enddo
```

CCE Runtime Activity Commentary

- `setenv CRAY_ACC_DEBUG <1,2,3>`

```
ACC: Initialize CUDA
ACC: Get Device 0
ACC: Create Context
ACC: Set Thread Context
ACC: Start transfer 2 items from saxpy.c:17
ACC:       allocate, copy to acc 'x' (4194304 bytes)
ACC:       allocate, copy to acc 'y' (4194304 bytes)
ACC: End transfer (to acc 8388608 bytes, to host 0 bytes)
ACC: Execute kernel saxpy$ck_L17_1 blocks:8192 threads:128 async(auto)
from saxpy.c:17
ACC: Wait async(auto) from saxpy.c:18
ACC: Start transfer 2 items from saxpy.c:18
ACC:       free 'x' (4194304 bytes)
ACC:       copy to host, free 'y' (4194304 bytes)
ACC: End transfer (to acc 0 bytes, to host 4194304 bytes)
```

PGI Runtime Commentary

- `setenv PGI_ACC_TIME 1`

```
Accelerator Kernel Timing data
/home/jlarkin/kernels/saxpy/saxpy.c
  saxpy NVIDIA devicenum=0
    time(us): 3,256
    11: data copyin reached 2 times
        device time(us): total=1,619 max=892 min=727 avg=809
    11: kernel launched 1 times
        grid: [4096] block: [256]
        device time(us): total=714 max=714 min=714 avg=714
        elapsed time(us): total=724 max=724 min=724 avg=724
    15: data copyout reached 1 times
          device time(us): total=923 max=923 min=923 avg=923
```

NVIDIA Profiling Tools

- **Command-line profiler available for OpenACC codes**
  - setenv COMPUTE_PROFILE 1
- **nvprof instruments your code to provide feedback on all CUDA-related activity**
  - Use nvprof to collect data for the CUDA Visual Profiler
- **NVIDIA Visual Profiler**
  - Offers performance observations

Vampir Toolsuite

- Trace-based performance analysis toolsuite
- VampirTrace provides instrumentation and performance data measurement
- Vampir provides a visual analysis of trace data
- Enhanced to support OpenACC

Cray Performance Tools – Example Profile

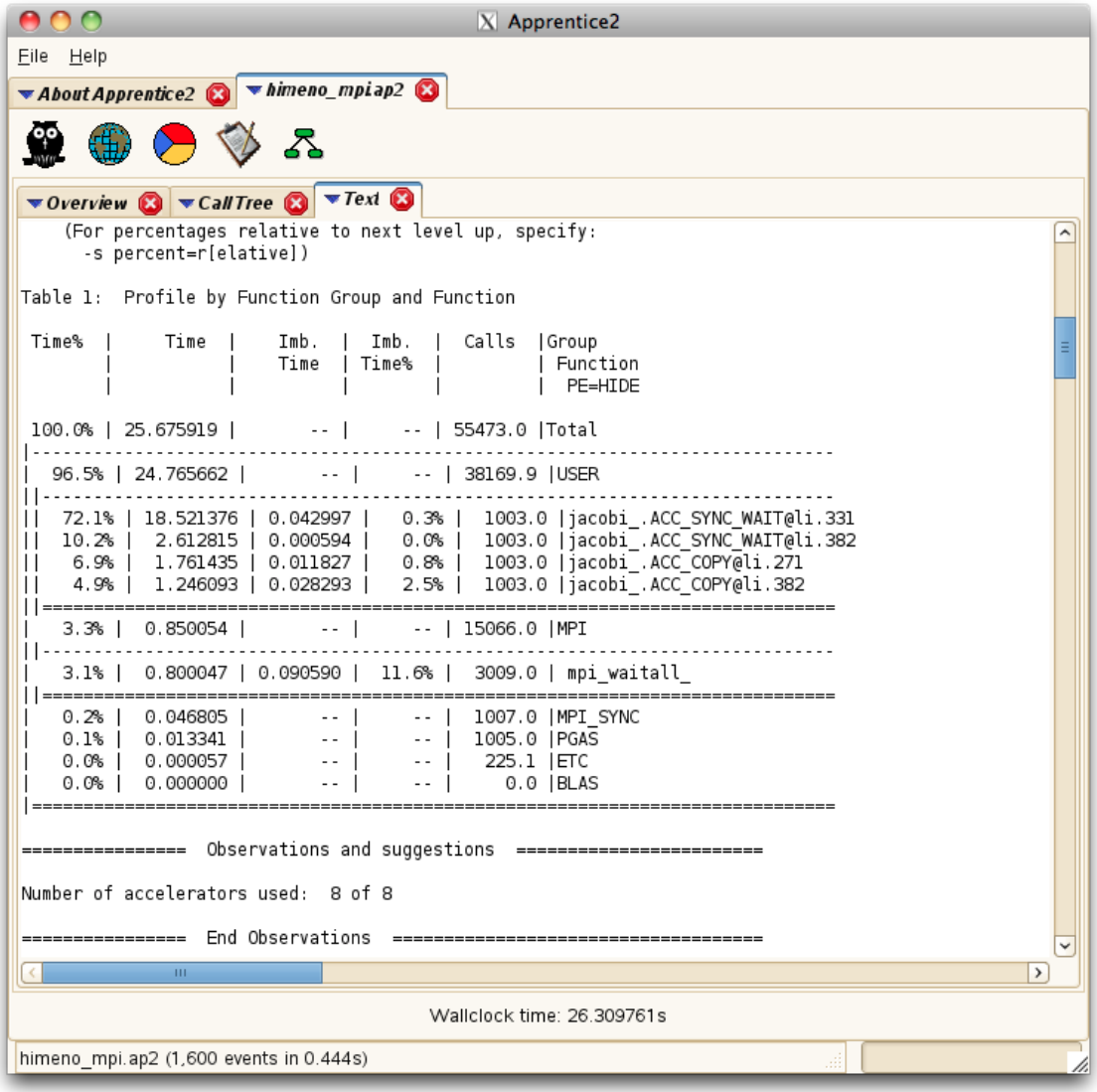
- Helpful for whole program analysis (MPI, GPU, I/O, etc)

```
Table 1:  Profile by Function

 Samp % |  Samp |  Imb.  |   Imb.  |Group
        |       |  Samp  | Samp %  | Function
        |       |        |         |  PE='HIDE'

 100.0% |  775  |    --  |     --  |Total
|-------------------------------------------
|  94.2% |  730  |    --  |     --  |USER
|-------------------------------------------
|  43.4% |  336  |  8.75  |    2.6% |mlwxyz_
|  16.1% |  125  |  6.28  |    4.9% |half_
|   8.0% |   62  |  6.25  |    9.5% |full_
|   6.8% |   53  |  1.88  |    3.5% |artv_
|   4.9% |   38  |  1.34  |    3.6% |bnd_
|   3.6% |   28  |  2.00  |    6.9% |currenf_
|   2.2% |   17  |  1.50  |    8.6% |bndsf_
|   1.7% |   13  |  1.97  |   13.5% |model_
|   1.4% |   11  |  1.53  |   12.2% |cfl_
|   1.3% |   10  |  0.75  |    7.0% |currenh_
|   1.0% |    8  |  5.28  |   41.9% |bndbo_
|   1.0% |    8  |  8.28  |   53.4% |bndto_
|===========================================
|   5.4% |   42  |    --  |     --  |MPI
|-------------------------------------------
|   1.9% |   15  |  4.62  |   23.9% |mpi_sendrecv_
|   1.8% |   14  | 16.53  |   55.0% |mpi_bcast_
|   1.7% |   13  |  5.66  |   30.7% |mpi_barrier_
|===========================================
```

Calltree from CrayPat (pat_report –O callers+src)

```
Table 1:  Profile by Function and Callers, with Line Numbers
 Time%  |      Time  |    Calls  |Group
        |            |           | Function
        |            |           |  Caller

100.0% | 14.548046 | 133356.0 |Total
|-------------------------------------------------------------------------
| 100.0% | 14.547826 | 133155.0 |USER
||------------------------------------------------------------------------
||  51.6% |  7.511687 |    191.0 |resid_.LOOPS
3||       |           |          | resid_:mg_v00.f:line.615
|||-----------------------------------------------------------------------
4|||  23.2% |  3.378418 |     21.0 |mg3p_.LOOPS:mg_v00.f:line.508
5|||       |           |          | mg3p_:mg_v00.f:line.473
||||----------------------------------------------------------------------
6|||| 22.1% |  3.217394 |     20.0 |mg_.LOOP.4.li.253:mg_v00.f:line.260
7||||       |           |          |  mg_.LOOPS:mg_v00.f:line.253
8||||       |           |          |   mg_:mg_v00.f:line.89
6||||  1.1% |  0.161024 |      1.0 |mg_.LOOPS:mg_v00.f:line.229
7||||       |           |          |  mg_:mg_v00.f:line.89
|||=======================================================================
4|||  22.2% |  3.227081 |     20.0 |mg_.LOOP.4.li.253:mg_v00.f:line.263
5|||       |           |          |  mg_.LOOPS:mg_v00.f:line.253
6|||       |           |          |   mg_:mg_v00.f:line.89
4|||   2.3% |  0.332700 |    147.0 |mg3p_.LOOP.2.li.486:mg_v00.f:line.501
5|||       |           |          |  mg3p_.LOOPS:mg_v00.f:line.486
6|||       |           |          |   mg3p_:mg_v00.f:line.473
7|||   2.2% |  0.316830 |    140.0 |    mg_.LOOP.4.li.253:mg_v00.f:line.260
8|||       |           |          |     mg_.LOOPS:mg_v00.f:line.253
9|||       |           |          |      mg_:mg_v00.f:line.89
4|||   1.7% |  0.250423 |      1.0 |mg_.LOOPS:mg_v00.f:line.221
5|||       |           |          |  mg_:mg_v00.f:line.89
4|||   1.1% |  0.161568 |      1.0 |mg_.LOOPS:mg_v00.f:line.248
5|||       |           |          |  mg_:mg_v00.f:line.89
4|||   1.1% |  0.161498 |      1.0 |mg_.LOOPS:mg_v00.f:line.229
5|||       |           |          |  mg_:mg_v00.f:line.89
|||=======================================================================
```

Profile with GPU Information



Example Report – Inclusive Loop Time

- Helpful when focusing on loops

Table 2: Loop Stats by Function (from -hprofile_generate)

Loop Incl Time Total	Loop Hit	Loop Trips Avg	Loop Trips Min	Loop Trips Max	Function=/.LOOP[.] PE=HIDE
8.995914	100	25	0	25	sweepy_.LOOP.1.li.33
8.995604	2500	25	0	25	sweepy_.LOOP.2.li.34
8.894750	50	25	0	25	sweepz_.LOOP.05.li.49
8.894637	1250	25	0	25	sweepz_.LOOP.06.li.50
4.420629	50	25	0	25	sweepx2_.LOOP.1.li.29
4.420536	1250	25	0	25	sweepx2_.LOOP.2.li.30
4.387534	50	25	0	25	sweepx1_.LOOP.1.li.29
4.387457	1250	25	0	25	sweepx1_.LOOP.2.li.30
2.523214	187500	107	0	107	riemann_.LOOP.2.li.63
1.541299	20062500	12	0	12	riemann_.LOOP.3.li.64
0.863656	1687500	104	0	108	parabola_.LOOP.6.li.67

Analyze Performance of Accelerated Program

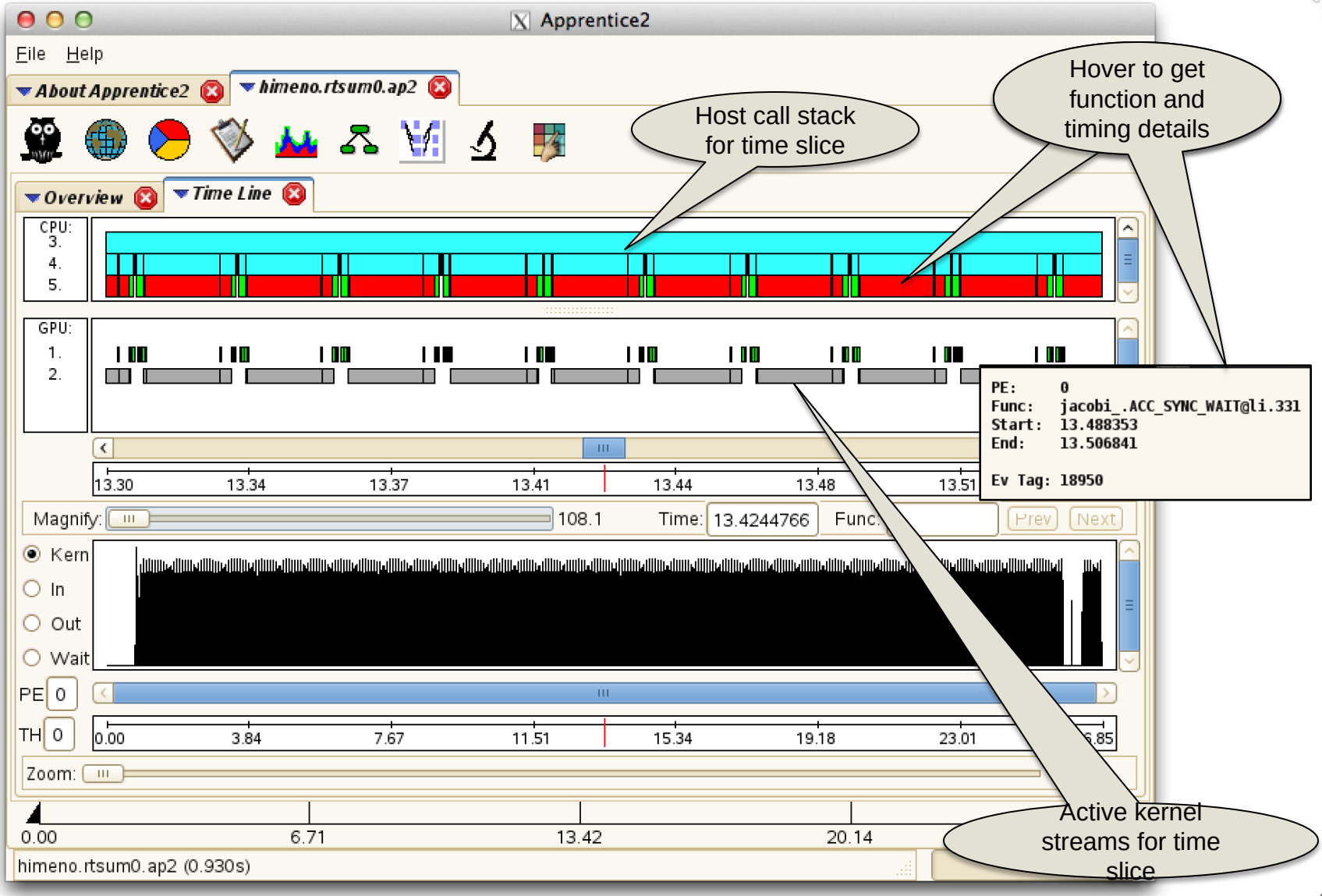
- **Statistics collected for programs with OpenACC directives**
  - Number of GPUs used in the job
  - Host time for kernel launches, data copies and synchronization with the accelerator
  - Accelerator time for kernel execution and data copies
  - Data copy size to and from the accelerator
  - Kernel grid size
  - Block size
  - Amount of shared memory dynamically allocated for kernel
  - GPU performance counters
  - Derived metrics based on performance counters

Example Accelerator Statistics

- Helpful when analyzing accelerated regions (data copies, execution time, etc.)

```
Table 1:  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% |   2.750  |   2.015  |  2812.760 |    13.568 |    103 |Total
|-------------------------------------------------------------------------------------------------------
| 100.0% |    2.750 |    2.015 |  2812.760 |    13.568 |    103 |lbm3d2p_d_
|        |          |          |           |           |        | lbm3d2p_d_.ACC_DATA_REGION@li.104
|||-----------------------------------------------------------------------------------------------------
3||  63.5% |    1.747 |    1.747 |  2799.192 |        -- |      1 |lbm3d2p_d_.ACC_COPY@li.104
3||  22.1% |    0.609 |    0.088 |    12.304 |    12.304 |     36 |streaming_
||||-----------------------------------------------------------------------------------------------------
4|||  20.6% |    0.566 |    0.046 |    12.304 |    12.304 |     27 |streaming_exchange_
5|||        |          |          |           |           |        | streaming_exchange_.ACC_DATA_REGION@li.526
6|||  18.8% |    0.517 |       -- |        -- |        -- |      1 |  streaming_exchange_.ACC_DATA_REGION@li.526(exclusive)
4|||   1.6% |    0.043 |    0.042 |        -- |        -- |      9 |streaming_.ACC_DATA_REGION@li.907
5|||   1.1% |    0.031 |    0.031 |        -- |        -- |      4 | streaming_.ACC_REGION@li.909
6|||   1.1% |    0.031 |       -- |        -- |        -- |      1 |  streaming_.ACC_REGION@li.909(exclusive)
||||=====================================================================================================

...
```

Overlap Between Host and Device



Porting a Larger Code

Adding OpenACC to a Larger Code

- **Adding OpenACC to a real code is not trivial work…**
  - Are parts of the program suitable for an accelerator?
  - Where do we start?
  - What do we do next?

- **We'll go through the exercise for an example code**
  - Running on a Cray XK7 (AMD Interlagos and Nvidia Kepler K20x)
  - Using Cray compiler and Cray performance analysis tools

The Code

- **NAS Parallel Benchmarks MG (MultiGrid) code**
  - Shorter than typical application
    - but structure of code is very similar
  - This example concentrates on the serial version
    - We also have parallel versions ported to OpenACC
    - The serial versions have OpenMP directives, but we do not use them during this exercise
  - Downloading it:
    - Fortran version: http://www.nas.nasa.gov/publications/npb.html.
      - 1445 lines, of which 267 blank
    - C version: http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp/download/download-benchmarks.html.
      - 1292 lines, of which 206 blank

Building and Running MG

- **Build:**
  - make MG [CLASS=<CLASS>] [<OPTIONS>]
    - CLASS is the problem size. "B" is the default.
  - Top-level Makefile passes options to MG/Makefile
    - this uses config/make.def for compiler-specific options

- **Run:**
  - Three important lines of output
    - Fortran
      - `L2 Norm is  0.1800564401355E-05`
      - `Mop/s total     =                1623.04`
      - `Verification    =              SUCCESSFUL`
    - C output same, but baseline performance differs
      - `Mop/s total     =                1320.26`
  - Always check:
    - L2 Norm should not be a NaN
    - Verification should be successful

Where Do We Start?

- Profile MG on the CPU

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

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

 100.0% | 12.069520 |    -- |     -- | 1630.0 |Total
|-------------------------------------------------------
| 100.0% | 12.069417 |    -- |     -- | 1230.0 |USER
||------------------------------------------------------
||  54.9% |  6.620529 |    -- |     -- |  161.0 |resid_
||  25.3% |  3.057070 |    -- |     -- |  160.0 |psinv_
||   9.5% |  1.148982 |    -- |     -- |  140.0 |rprj3_
||   8.1% |  0.983395 |    -- |     -- |  140.0 |interp_
||   1.3% |  0.153775 |    -- |     -- |  461.0 |comm3_
|=======================================================
```

- Four routines dominate the runtime
  - More than half the time is spent in **resid**
  - There are other routines executing for less than 1% of the total time
    - These might be important for the OpenACC port

Understand Flow of the Application

```
Table 1:  Function Calltree View

 Time%  |     Time  | Calls  |Calltree

 100.0% | 12.069520 | 1630.0 |Total
|-------------------------------------
| 100.0% | 12.069417 | 1230.0 |mg_
||------------------------------------
||  72.3% |  8.724588 | 1180.0 |mg3p_
|||-----------------------------------
3||  28.3% |  3.416675 |  280.0 |resid_
3||  25.8% |  3.108020 |  320.0 |psinv_
3||   9.6% |  1.160157 |  280.0 |rprj3_
3||   8.1% |  0.983395 |  140.0 |interp_
|||====================================
||  27.3% |  3.295504 |   42.0 |resid_
|=====================================
```

- **mg calls:**
  - mg3p (which then calls resid, psinv, rprj3, interp)
  - resid also called directly from mg

Get Work Estimates for Loops

```
Table 2:  Loop Stats by Function (from -hprofile_generate)

    Loop  |   Loop  |   Loop  |  Loop  |  Loop  |Function=/.LOOP[.]
    Incl  |    Hit  |  Trips  | Trips  | Trips  |
    Time  |         |    Avg  |   Min  |   Max  |
   Total  |         |         |        |        |
|-------------------------------------------------------------------
| 6.830878 |     161 |  96.497 |      4 |    256 |resid_.LOOP.1.li.634
| 6.830032 |   15536 | 201.067 |      4 |    256 |resid_.LOOP.2.li.635
| 4.033780 | 3123776 | 237.548 |      6 |    258 |resid_.LOOP.3.li.636
| 2.607888 | 3123776 | 235.548 |      4 |    256 |resid_.LOOP.4.li.642
```

- **Loop-level profiling is more useful now**
  - Which loopnests (rather than just routines) took most time?
  - How may iterations did this loopnest have?
- **Here are the lines relating to resid**
  - Loops starting at 636 and 642 are nested inside loops at line 634, 635
    - See how the Loop Hit numbers multiply up
    - See how inclusive times for 636 and 642 add to give that for 635
    - Inclusive times for 634, 635 same: perfectly nested loops

Add First OpenACC Kernel

- **Clearly we should start with resid**
  - Fortran:
    - !$acc parallel loop vector_length(NTHREADS)
      !$acc& private(u1,u2) copyin(u,v,a) copyout(r)
  - C:
    - #pragma acc parallel loop vector_length(NTHREADS) \
      private(u1,u2) copyin(u[0:n1*n2*n3],v[0:n1*n2*n3],a[0:4]) \
      copyout(r[0:n1*n2*n3])
    - Data movement sizes explicit to avoid "unshaped pointer" errors
      - Because we are dynamically allocating memory

Resulting MG Performance?

- Running with and without OpenACC kernel:

	Original (Mop/s)	1 kernel (Mop/s)
Fortran	1623.04	1541.42
C	1320.26	1409.48

- So the code is actually slower... **Why?**

Enable Cray Runtime Commentary

- export CRAY_ACC_DEBUG=2
- for every call to resid:

```
ACC: Start transfer 6 items from mg_v03.f:615
ACC:       allocate, copy to acc 'a' (32 bytes)
ACC:       allocate 'r' (137388096 bytes)
ACC:       allocate, copy to acc 'u' (137388096 bytes)
ACC:       allocate, copy to acc 'v' (137388096 bytes)
ACC:       allocate <internal> (530432 bytes)
ACC:       allocate <internal> (530432 bytes)
ACC: End transfer (to acc 274776224 bytes, to host 0 bytes)
ACC: Execute kernel resid_$ck_L615_1 blocks:256 threads:128 async(auto) from mg_v03.f:615
ACC: Wait async(auto) from mg_v03.f:639
ACC: Start transfer 6 items from mg_v03.f:639
ACC:       free 'a' (32 bytes)
ACC:       copy to host, free 'r' (137388096 bytes)
ACC:       free 'u' (137388096 bytes)
ACC:       free 'v' (137388096 bytes)
ACC:       free <internal> (0 bytes)
ACC:       free <internal> (0 bytes)
ACC: End transfer (to acc 0 bytes, to host 137388096 bytes)
```

- Certainly a lot of data was moved
  - Commentary tells us which arrays, at which line and how much data

Or Use Nvidia Compute Profiler

- **export COMPUTE_PROFILE=1**
  - Analyses PTX (from OpenACC or from CUDA)
  - Very useful if mixing OpenACC with CUDA code

```
method=[ memcpyHtoD ] gputime=[ 1.088 ] cputime=[ 42.000 ]
method=[ memcpyHtoD ] gputime=[ 52236.543 ] cputime=[ 52513.000 ]
method=[ memcpyHtoD ] gputime=[ 52153.281 ] cputime=[ 52402.000 ]
method=[ resid_$ck_L615_1 ] gputime=[ 15063.424 ] cputime=[ 21.000 ] occupancy=[ 0.333 ]
method=[ memcpyDtoH ] gputime=[ 281508.594 ] cputime=[ 283700.000 ]
```

- **Data transfers obvious, taking most time**

Or Use CrayPAT for a Profile by Function

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

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

 100.0% | 16.409900 |    -- |     -- | 1252.0 |Total
|-------------------------------------------------------------------------
| 100.0% | 16.409731 |    -- |     -- |  851.0 |USER
||------------------------------------------------------------------------
||  51.2% |  8.403343 |    -- |     -- |    1.0 |mg_
||  34.3% |  5.622111 |    -- |     -- |  170.0 |resid_.ACC_COPY@li.615
||  11.8% |  1.936478 |    -- |     -- |  170.0 |resid_.ACC_COPY@li.639
||   2.7% |  0.440894 |    -- |     -- |  170.0 |resid_.ACC_SYNC_WAIT@li.639
||   0.0% |  0.005727 |    -- |     -- |  170.0 |resid_.ACC_KERNEL@li.615
||   0.0% |  0.001178 |    -- |     -- |  170.0 |resid_.ACC_REGION@li.615
||========================================================================
|   0.0% |  0.000170 |    -- |     -- |  401.0 |ETC
||------------------------------------------------------------------------
```

- **Provides aggregated report of data movements**
  - names, sizes and frequencies of original arrays lost
- **Shows asynchronous kernel launches**
  - Notice ACC_KERNEL almost zero
  - SYNC_WAIT shows the compute time
  - could recompile with -hacc_model=auto_async_none

…And CrayPAT Accelerator Statistics

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

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

 100.0% | 8.007 | 7.962 |     12341 |      6171 |     850 |Total
|-------------------------------------------------------------------------------------------
| 100.0% | 8.007 | 7.962 |     12341 |      6171 |     850 |mg_
||------------------------------------------------------------------------------------------
||  50.0% | 4.005 | 3.969 |      6314 |      3157 |     735 |mg3p_
3|        |       |       |           |           |         | resid_
4|        |       |       |           |           |         |  resid_.ACC_REGION@li.615
|||||----------------------------------------------------------------------------------------
5||||  36.2% | 2.898 | 2.877 |      6314 |        -- |     147 |resid_.ACC_COPY@li.615
5||||  10.8% | 0.867 | 0.860 |        -- |      3157 |     147 |resid_.ACC_COPY@li.639
5||||   2.9% | 0.235 |    -- |        -- |        -- |     147 |resid_.ACC_SYNC_WAIT@li.639
5||||   0.1% | 0.004 | 0.232 |        -- |        -- |     147 |resid_.ACC_KERNEL@li.615
5||||   0.0% | 0.001 |    -- |        -- |        -- |     147 |resid_.ACC_REGION@li.615(exclusive)
```

- **Host and accelerator times given separately**
  - ACC_KERNEL
    - Acc Time is the compute time
    - Host Time is the time for the asynchronous launch
      - The Host "catches up" at the SYNC_WAIT

… And CrayPAT Summarized Trace

- pat_build -u mg.B.x

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

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

 100.0% | 16.452925 |    -- |     -- | 265303.0 |Total
|-------------------------------------------------------------------------
| 100.0% | 16.452760 |    -- |     -- | 264902.0 |USER
||------------------------------------------------------------------------
||  34.2% |  5.621172 |    -- |     -- |    170.0 |resid_.ACC_COPY@li.615
||  19.4% |  3.199216 |    -- |     -- |    168.0 |psinv_
||  11.8% |  1.940111 |    -- |     -- |    170.0 |resid_.ACC_COPY@li.639
||  10.7% |  1.764268 |    -- |     -- | 131072.0 |vranlc_
||   7.4% |  1.217534 |    -- |     -- |    147.0 |rprj3_
||   6.3% |  1.033920 |    -- |     -- |    147.0 |interp_
||   4.3% |  0.709337 |    -- |     -- |    151.0 |zero3_
||   2.7% |  0.441237 |    -- |     -- |    170.0 |resid_.ACC_SYNC_WAIT@li.639
||   1.5% |  0.240856 |    -- |     -- |      2.0 |zran3_
||   1.0% |  0.170554 |    -- |     -- |    487.0 |comm3_
|=========================================================================
```

- resid kernel no longer dominates the profile
  - actual compute time is shown in SYNC_WAIT (Host) Time
  - Its data copies are significant, however

More OpenACC Kernels

- Running with 4 accelerated kernels:

	Original (Mop/s)	1 kernel (Mop/s)	4 kernels (Mop/s)
Fortran	1623.04	1541.42	1274.48
C	1320.26	1409.48	991.42

- Even slower, and C particularly bad. **Why?**

Profile the Code Again

- Notice that spending most time in interp
- Next look at compiler listing:

```
727.              #pragma acc parallel loop private(z1,z2,z3)  \
728.                copy(u[0:n1*n2*n3])                        \
729.                copyin(z[0:mm1*mm2*mm3])
730.  gG------<>       for (i3 = 0; i3 < mm3-1; i3++) {
731.  1--------<             for (i2 = 0; i2 < mm2-1; i2++) {
732.  1 g------<              for (i1 = 0; i1 < mm1; i1++) {
733.  1 g                       i123 = i1 + mm1*i2 + mm12*i3;
734.  1 g                         z1[i1] = z[i123+mm1] + z[i123];
735.  1 g                         z2[i1] = z[i123+mm12] + z[i123];
736.  1 g                         z3[i1] = z[i123+mm1+mm12] + z[i123+mm12] + z1[i1];
737.  1 g------>              }
738.  1 r4-----<              for (i1 = 0; i1 < mm1-1; i1++) {
739.  1 r4                      i123 = i1 + mm1*i2 + mm12*i3;
740.  1 r4                      j123 = 2*i1 + n1*(2*i2 + n2 * 2*i3);
741.  1 r4                        u[j123] += z[i123];
742.  1 r4                        u[j123+1] += 0.5*(z[i123+1]+z[i123]);
743.  1 r4----->              }
```

Loop Accelerated

Loop Partitioned

Loop Not Partitioned

- Loop at line 738 not partitioned
  - Executed redundantly: every thread does every loop iteration

More OpenACC Kernels

- **Insert directive above unpartitioned loop to help compiler:**
  - #pragma acc loop independent
  - (if this didn't work, would use "#pragma acc loop vector" instead

	Original (Mop/s)	1 kernel (Mop/s)	4 kernels (Mop/s)
Fortran	1623.04	1541.42	1274.48
C	1320.26	1409.48	1271.12

- **Fortran and C performance now identical**

Next Steps – Reduce Data Movement

- Need to introduce data regions higher up calltree
- For this, need all callee routines to be accelerated with OpenACC directives
- Use a profiler to map out the calltree to get list of routines
- First need to port some insignificant routines
  - norm2u3, zero3, comm3

Results for Accelerating up the Calltree

- Accelerated loops in norm2u3, zero3, comm3

	Original (Mop/s)	1 kernel (Mop/s)	4 kernels (Mop/s)	Calltree Routines (Mop/s)
Fortran	1623.04	1541.42	1274.48	886.02
C	1320.26	1409.48	1271.12	873.01

- Slower because even more data movement
  - C still slightly down; poor scheduling needs acc loop independent

	Original (Mop/s)	1 kernel (Mop/s)	4 kernels (Mop/s)	Calltree Routines (Mop/s)
Fortran	1623.04	1541.42	1274.48	886.02
C	1320.26	1409.48	1271.12	885.27

Add Data Region

- **Now we put a data region in the main program**
  - Arrays u,v,r are declared create
    - We'll never use the host version of these
  - Arrays a,c are declared copyin
    - They're initialized on the host

- **Then in all the subprograms, we change clauses**
  - Replace copy* and create by present
    - Could replace by present_or_*
      - If we know they should always be present, better to state this
      - Then mistakes become runtime errors rather than just wrong answers
        - We'd have to diagnose these by trawling the runtime commentary

Results with Data Region in Main

- At last, we are running faster (and correctly)!

	Original (Mop/s)	1 kernel (Mop/s)	4 kernels (Mop/s)	Calltree Routines (Mop/s)	Data Region (Mop/s)
Fortran	1623.04	1541.42	1274.48	886.02	23913.21
C	1320.26	1409.48	1271.12	885.27	23558.03

View Compiler Commentary Again

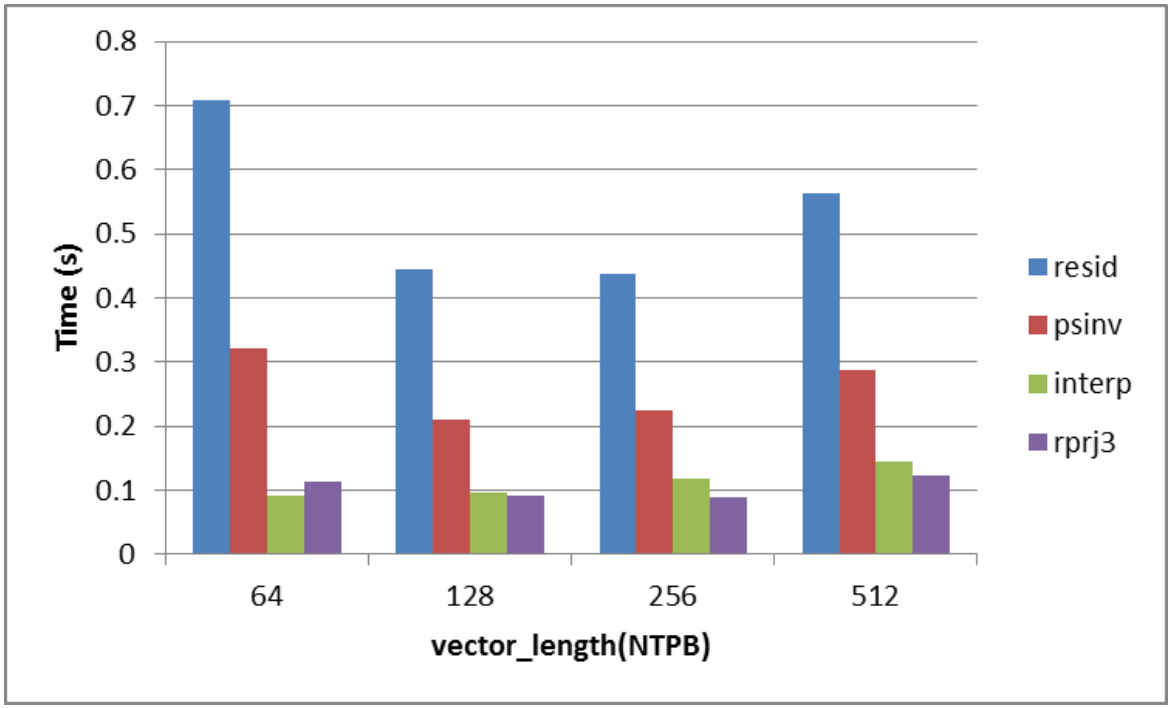
- **All array data transfers eliminated**
  - Run with CRAY_ACC_DEBUG=2 and catch STDERR in a file
  - grep "copy" <file> | sort | uniq

```
ACC:       allocate, copy to acc 'a' (32 bytes)
ACC:       allocate, copy to acc 'c' (32 bytes)
ACC:       allocate, copy to acc 'jg' (320 bytes)
ACC:       allocate reusable, copy to acc <internal> (16 bytes)
ACC:       allocate reusable, copy to acc <internal> (4 bytes)
ACC:       copy to host, done reusable <internal> (16 bytes)
ACC:       reusable acquired, copy to acc <internal> (16 bytes)
```

  - Arrays a, c, jg copied only at initialization
  - Some internal transfers unavoidable

Performance Tuning Tips

- **Check the .lst loopmark file**
  - Are any kernels obviously badly scheduled?
    - No, we already checked that
- **Try varying vector_length from the default of 128**
  - Different values may suit different kernels
  - Effect small here: up to 5% per kernel, but only 2% overall:



Performance Tuning Tips

- **Loop scheduling**
  - Collapse loops within loopnests
    - OpenACC schedules according to the loops in the nest
    - Collapsing loops can increase the tripcount
      - e.g. to allow more threads in a block
  - Using the worker clause may help for imperfectly nested loops
  - Block loops in a loopnest
    - This can improve cache usage (as on the CPU)
    - CCE-specific directives can do this, or try it manually
- **More extreme:**
  - Avoid temporary arrays
    - Use private scalars, as these are more likely to go into registers
  - Rewrite the most expensive kernels in CUDA and handtune
    - but remember you are competing against a whole compiler team

Other Performance Tuning Improvements

- **Avoiding temporary arrays in resid:**
  - `Mop/s total = 23999.08 ! Fortran`
  - `Mop/s total = 23640.24 // C`
  - benefit v. small; was it really worth hacking the source?
- **Call an external CUDA version of resid**
  - `Mop/s total = 22351.51 ! Fortran`
  - `Mop/s total = 21289.72 // C`
  - This was a naive kernel
    - (Even so, there may be a lesson in this)
- **Data movement was a far bigger optimisation**
  - than any of our kernel improvements

Conclusions: How far did we get?

- **Significant speedup compared to single core:**
  - Fortran: 1826.19 → 23999.08 = 15x
  - C: 1320.26 → 23640.24 = 18x
- **The real comparison is to a full CPU or node**
  - run the OpenMP version of the code
    - across 16 cores for an XK6 node (single AMD Interlagos)
      - `Mop/s total = 9162.37 ! Fortran, Cray XK7 CPU, 16 threads`
      - `Mop/s total = 8638.68 // C, Cray XK7 CPU, 16 threads`
    - across 32 cores for an XE6 node (dual AMD Interlagos)
      - `Mop/s total = 15244.09 ! Fortran, Cray XE6, 32 threads`
      - `Mop/s total = 15120.86 // C, Cray XE6, 32 threads`
  - maybe MPI version would scale better, but our code here is scalar
- **Final speedup compared to full node:**
  - XK7 CPU node (16 cores): **2.6x**
  - XE6 CPU node (32 cores): **1.5x**
    - This is for Fortran, C looks slightly better

Conclusions: How much further could we get?

- **So we are 2.6x or 1.5x faster, node-for-node**
  - How much faster could we get (speeds and feeds)
    - Flops and mem. b/w around 5-10x faster than single AMD Interlagos
- **So why were we not faster?**
  - MultiGrid application cycles through grid sizes
    - sometimes the grid is really small: 4x4x4
    - CrayPAT loop profiling showed us that
  - Small grid sizes will never schedule well on the GPU
    - consider checking grid size and only using OpenACC for larger ones
    - or do we even need the smaller grids?

Advanced Topics

Contents

- **Some more advanced OpenACC topics**
  - Advanced data management
    - enter/exit data constructs
    - declare directive
    - Some “examples”
  - Separate compilation support
    - routine construct
  - Device specailization
    - device_type clause
  - nested parallelism
  - runtime routines
  - Interoperability feature
    - host_data construct
    - CUDA example

Unstructured data lifetimes

Enter Data construct

- In C and C++, the syntax is
  - #pragma acc enter data *clause-list new-line*
- In Fortran, the syntax is
  - !$acc enter data *clause-list*
- *clause* :
  - if(*condition*)
  - async *[(int-expr)]*
  - wait *[(int-expr-list)]*
  - copyin(*var-list*)
  - create(*var-list*)
  - present_or_copyin(*var-list*)
  - present_or_create(*var-list*)

Exit Data construct

- In C and C++, the syntax is
  - #pragma acc exit data *clause-list new-line*
- In Fortran, the syntax is
  - !$acc enter data *clause-list*
- *clause* :
  - if(*condition*)
  - async *[(int-expr)]*
  - wait *[(int-expr-list)]*
  - copyout(var-list)
  - delete(var-list)

Global Object management

- **!$acc declare**
  - Makes a variable resident in accelerator memory
    - persists for the duration of the implicit data region
  - Makes a variable available in accelerator code for linking
- **clauses**
  - device_resident(obj-list)
    - Place object on device as a global, only device copy is required
  - link(obj-list)
    - Place a pointer in device global memory for the listed objects
    - Programmer is responsible for actually moving object to the device

Data management features and global data

```
float a[1000000];
#pragma acc declare create(a )

extern float a[];
#pragma acc declare create(a)
```

```
float a[100000];
#pragma acc declare device_resident(a)

float a[100000];
#pragma acc declare link(a)

float *a;
#pragma acc declare create(a)
```

Data management features
unstructured data lifetimes

```
#pragma acc data copyin(a[0:n])\
                 create(b[0:n])
{ … }
```

```
#pragma acc enter data copyin( a[0:n] )\
             create(b[0:n])
…
#pragma acc exit data delete(a[0:n])
 …
#pragma acc exit data copyout(b[0:n])
```

```
void init() {
#pragma acc enter data copyin( a[0:n] )\
                 create(b[0:n])
}

void fini {
#pragma acc exit data delete(a[0:n])
#pragma acc exit data copyout(b[0:n])
}
```

Procedure calls, separate compilation

- In C and C++, the syntax of the **routine** directive is:
  - #pragma acc routine *clause-list new-line*
  - #pragma acc routine (*name*) *clause-list new-line*
- In Fortran the syntax of the **routine** directive is:
  - !$acc routine *clause-list*
  - !$acc routine (*name*) *clause-list*
- The *clause* is one of the following:
  - gang
  - worker
  - vector
  - seq
  - bind(*name*)
  - bind(*string*)
  - device_type(*device-type-list*)
  - nohost

Routine directive examples

```
#pragma acc routine gang bind( foo_nvidia )
void foo(int *a, int n) {
#pragma acc loop gang vector
 for( int i = 0; i<n; i++)
  a(i) = i
}
```

```
subroutine foo(a, n)
!$acc routine gang bind( foo_nvidia )
integer,dimension(:) :: a
integer :: i
!$acc loop gang vector
 do i = 1, n
  a(i) = i
 end do
end subroutine
```

Device-specific tuning, multiple devices

- device_type(dev-type)

```
#pragma acc parallel loop \
            device_type(nvidia) num_gangs(200) …\
            dtype(radeon) num_gangs(400) …
 for( int i = 0; i < n; ++i ){
    v[i] += rhs[i];
    matvec( v, x, a, i, n );
 }
```

Nested Parallelism

- Actually simply a deletion of two restrictions
  - OpenACC parallel regions may not contain other parallel regions or kernels regions.
  - OpenACC kernels regions may not contain other parallel regions or kernels regions.
- Other changes were mainly cosmetic
- Has significant impact on where objects can be placed in memory.

The OpenACC Runtime API

The OpenACC runtime API

- **Directives are comments in the code**
  - automatically ignored by non-accelerating compiler
- **OpenACC also offers a runtime API**
  - set of library calls, names starting acc_
    - set, get and control accelerator properties
    - offer finer-grained control of asynchronicity
    - Data allocation and movement
  - OpenACC specific
    - will need pre-processing away for CPU execution
    - #ifdef _OPENACC
- **CCE offers an extended runtime API**
  - V2 adopted almost all
  - set of library calls, names starting with cray_acc_
    - will need preprocessing away if not using OpenACC with CCE
    - #if defined(_OPENACC) && PE_ENV==CRAY
- **Advice: you do not need the API for most codes.**
  - Start without it, only introduce it where it is really needed.
  - I almost never use it; we'll talk about it later.
  - !$acc enter data and !$acc exit data are alternatives

Runtime API for Device Selection and Control

- **About the OpenACC-supporting accelerators**
  - What type of device will I use next? acc_get_device_type()
    - default from environment variable ACC_DEVICE_TYPE
  - What type of device should I use next? acc_set_device_type()

  - How many accelerators of specified type? acc_get_num_devices()
  - Which device of specified type will I use next? acc_get_device_num()
    - default from environment variable ACC_DEVICE_NUM
  - Which device of specified type should I use next? acc_set_device_num()

  - Am I executing on device of specified type? acc_on_device()

- **Initialising/shutting down accelerators:**
  - Initialise (e.g. to isolate time taken): acc_init()
  - Shut down (e.g. before switching devices): acc_shutdown()

Runtime API for Advanced Memory Control

- These are for very advanced users
- Offer method to allocate and free device memory
- C/C++ only:
  - void* acc_malloc (size_t);
  - void acc_free (void*);
- All languages (C/C++ shown)
  - acc_copyout(void*, size_t);
  - acc_delete(void*, size_t);
  - acc_is_present(void*, size_t);
  - acc_update_device(void*, size_t);
  - acc_update_self(void*, size_t);
  - acc_[present_or_]copyin(void*, size_t);
  - acc_[present_or_]create(void*, size_t);

- If you just need to know the address of the memory used by OpenACC (to pass, for instance, to CUDA)
  - then you don't need these
  - just use host_data directive or acc_deviceptr instead

Fortran advanced memory control example

- Specification description

```
subroutine acc_copyin( a )
  type, dimension(:[,:]…) :: a

subroutine acc_copyin( a, len )
  type :: a
  integer :: len
```

- Fortran 2008 module file declaration

```
interface acc_copyin
  subroutine acc_copyin_1 (a)
    type(*),dimension(..) :: a
  end subroutine acc_copyin_1
  subroutine acc_copyin_2 (a, len)
    type(*),dimension(*) :: a
    integer,value :: len
  end subroutine acc_copyin_2
end interface
```

Advanced Memory management

- Present table manipulation routines
  - acc_map_data(dvoid*, hvoid*, size_t);
    - Register device memory in present table as “shadowing” the given host memory.
  - acc_unmap_data(hvoid*);
    - Remove present table relationship for object.
- Memory movement functions
  - acc_memcpy_to_device(dvoid*, hvoid*, size_t);
  - acc_memcpy_from_device(dvoid*, hvoid*, size_t);
  - Cray provides async versions of these routines
    - cray_acc_memcpy_to_device_async(dvoid*, hvoid*, size_t, int);
    - acc_memcpy_from_device_async(dvoid*, hvoid*, size_t, int)

Runtime API for Asynchronicity

- **Runtime API can be used to control asynchronicity**
  - Advice: this is probably the part of the API you are most likely to use in fortran
- **Waiting for stream of operations to complete**
  - acc_async_wait(handle)
    - duplicates functionality of !$acc wait(handle) directive
  - acc[_async]_wait_async(handle1, handle2)
    - duplicates functionality of !$acc wait(handle1) async(handle2) directive
- **Waiting for all operations to complete**
  - acc_async_wait_all()
    - duplicates functionality of !$acc wait directive
  - acc[_async_]wait_all_async(handle)
    - duplicates functionality of !$acc wait async(handle) directive
- **Can also test for completion without waiting**
  - a single stream of operations: acc_async_test(handle)
  - all operations: acc_async_test_all()
  - no directive equivalent for these

host_data directive

- **OpenACC runtime manages GPU memory implicitly**
  - user does not need to worry about memory allocation/free-ing
- **Sometimes it can be useful to know where data is held in device memory, e.g.:**
  - so a hand-optimised CUDA kernel can be used to process data already held on the device
  - so a third-party GPU library can be used to process data already held on the device (Cray libsci_acc, cuBLAS, cuFFT etc.)
  - so optimised communication libraries can be used to streamline data transfer from one GPU to another
- **host_data directive provides mechanism for this**
  - nested inside OpenACC data region
  - subprogram calls within host_data region then pass pointer in device memory rather than in host memory
- **acc_deviceptr(hvoid*)**
- **acc_hostptr(dvoid*)**

Interoperability with CUDA

```
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 before we return to the OpenACC
  - CUDA kernel written as usual
  - Or use same mechanism to call existing CUDA library

Summary

- **You have now seen everything in the OpenACC 2.0 spec.**
  - At least from a 20,000 foot level

- **OpenACC is?**
  - Powerful
  - Robust
  - Scary
  - Hard
  - …

Debugging OpenACC Applications with DDT or TotalView

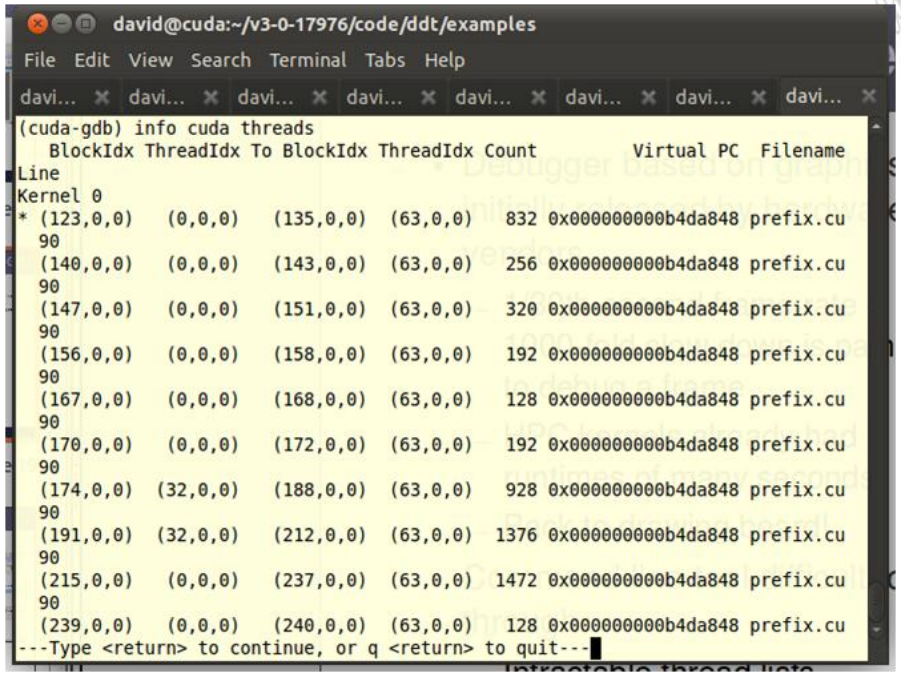
Courtesy from

David Lecomber
Allinea Software

Chris Gottbrath
Rogue Wave Software

How Do We Fix GPU Bugs?

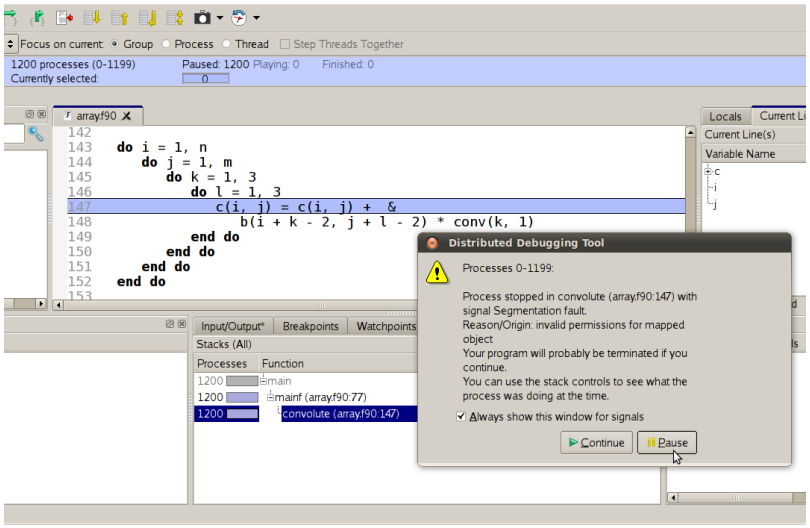
- **Print statements**
  - Too intrusive
- **Command line debugger?**
  - A good start:
    - Variables, source code
    - Large thread counts overwhelming
  - Too complex
- **A graphical debugger...**

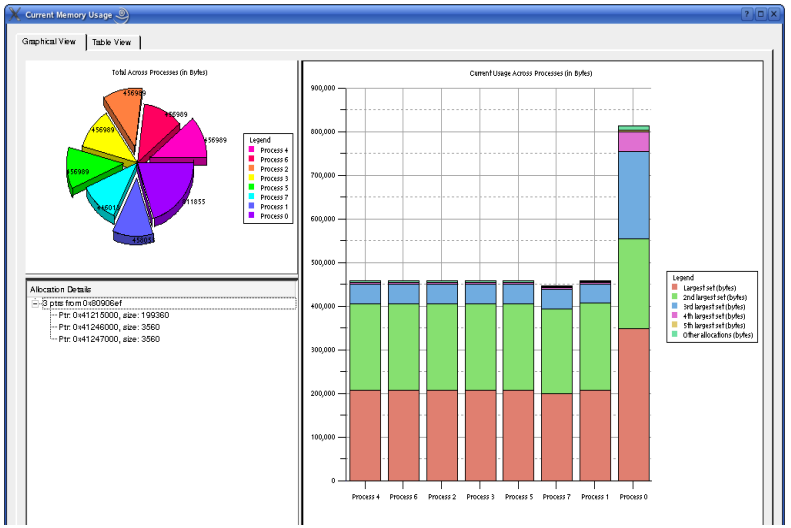


Allinea DDT in a Nutshell

allinea
www.allinea.com

- **Graphical source level debugger for**
  - Parallel, multi-threaded, scalar or hybrid code
  - C, C++, F90, Co-Array Fortran, UPC
- **Strong feature set**
  - Memory debugging
  - Data analysis
- **Managing concurrency**
  - Emphasizing differences
  - Collective control

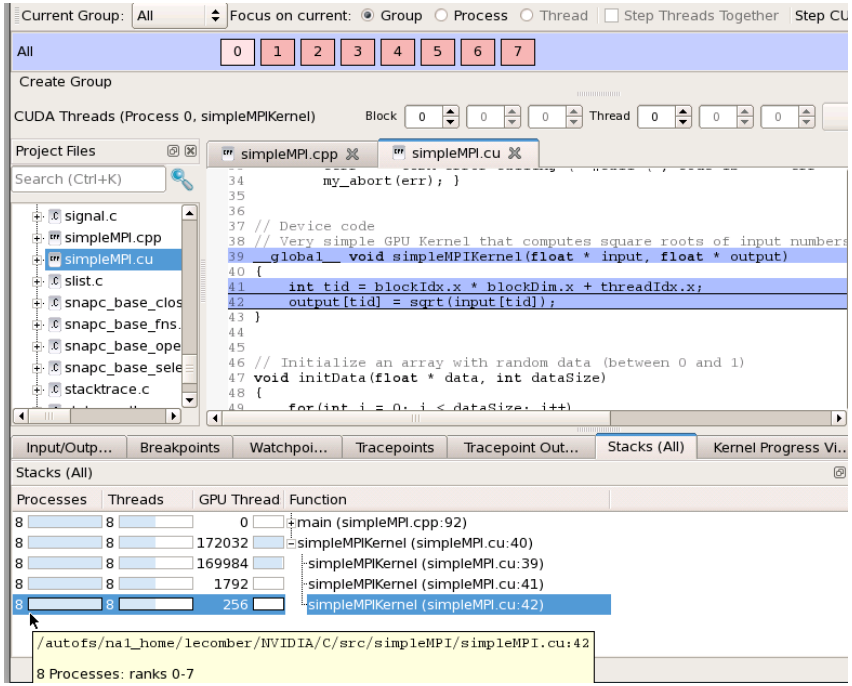




GPU Debugging with Allinea DDT

allinea
www.allinea.com

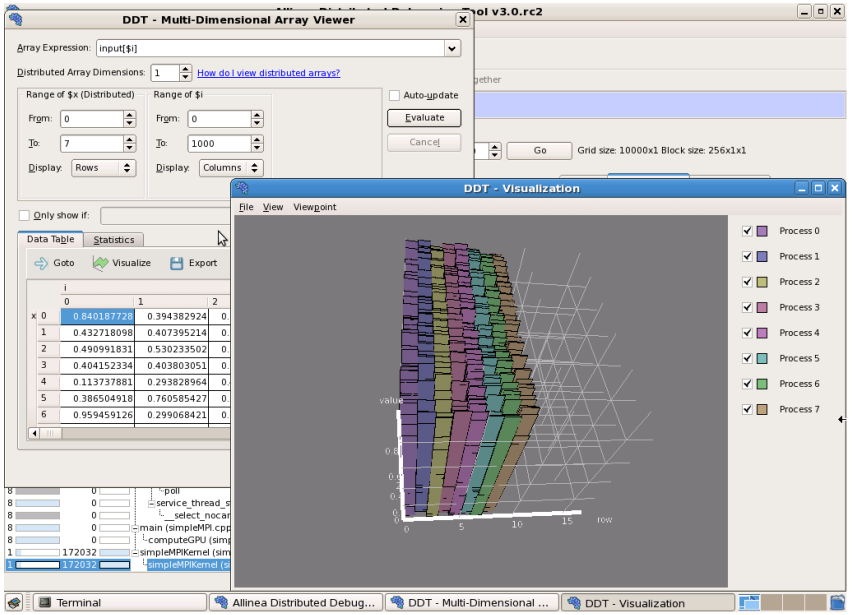
- **Almost like debugging a CPU – you can still:**
  - Run through to a crash
  - Step through and observe
- **CPU-like debugging features**
  - Double click to set breakpoints
  - Hover the mouse for more information
  - Step a warp, block or kernel
  - Follow threads through the kernel
- **Simultaneously debugs CPU code**
- **CUDA Memcheck feature detects read/write errors**



Examining GPU data

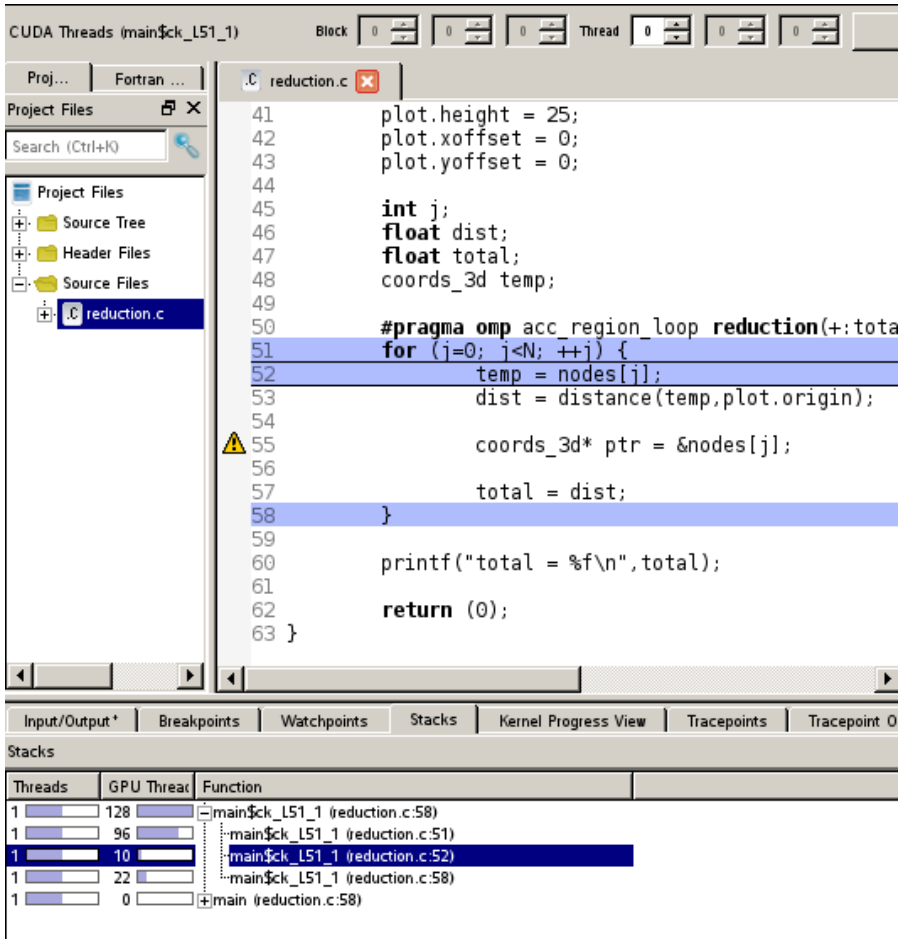
allinea
www.allinea.com

- **Debugger reads host and device memory**
  - Shows all memory classes: shared, constant, local, global, register..
  - Able to examine variables
  - … or plot larger arrays directly from device memory



OpenACC Debugging

allinea
www.allinea.com



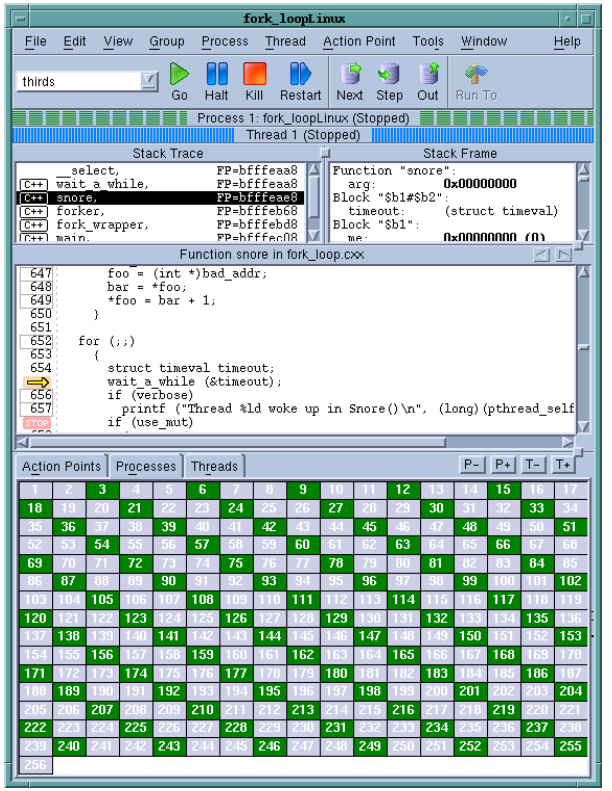
- **On device debugging with Allinea DDT**
  - Variables – arrays, pointers, full F90 and C support
  - Set breakpoints and step warps and blocks
- **Requires Cray compiler for on-device debugging**
  - Other compilers to follow
- **Identical to CUDA**
  - Full warp/block/kernel controls
- **Some DDT/OpenACC users**
  - ORNL – Titan (18,688 GPUs)
  - NCSA - Blue Waters (3K+ GPUs)
  - CSCS

What is TotalView?

ROGUE WAVE SOFTWARE

- **Application Analysis and Debugging Tool: Code Confidently**
  - Debug and Analyze C/C++ and Fortran on Linux, Unix or Mac OS X
  - Laptops to supercomputers such as the Cray XE or XK
  - Makes developing, maintaining and supporting critical apps easier and less risky

- **Major Features include**
  - Easy to learn graphical user interface with data visualization
  - Parallel debugging with
    - MPI
    - OpenMP
  - Accelerator debugging of apps that use
    - CUDA
    - OpenACC
  - Memory Debugging with MemoryScape
  - Deterministic Replay Capability Included on Linux/x86-64

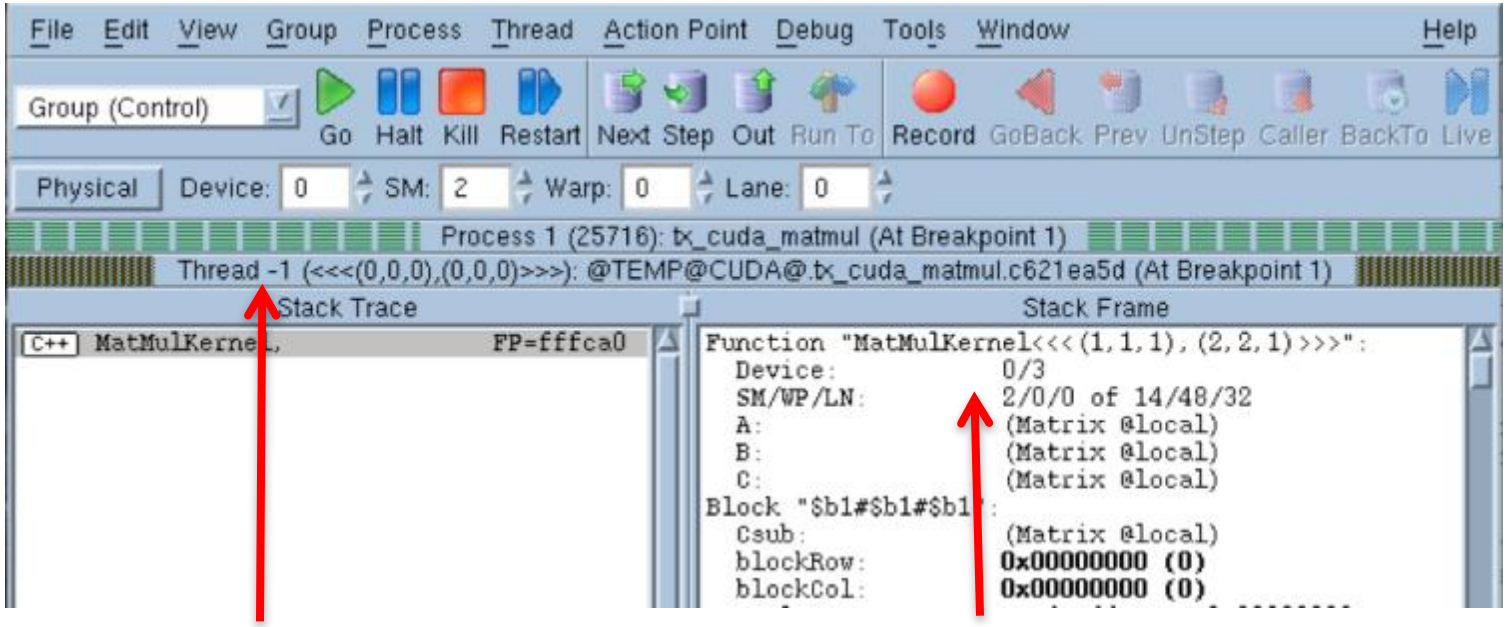


Debugging CUDA with Totalview

ROGUE WAVE SOFTWARE

Navigate freely through your code

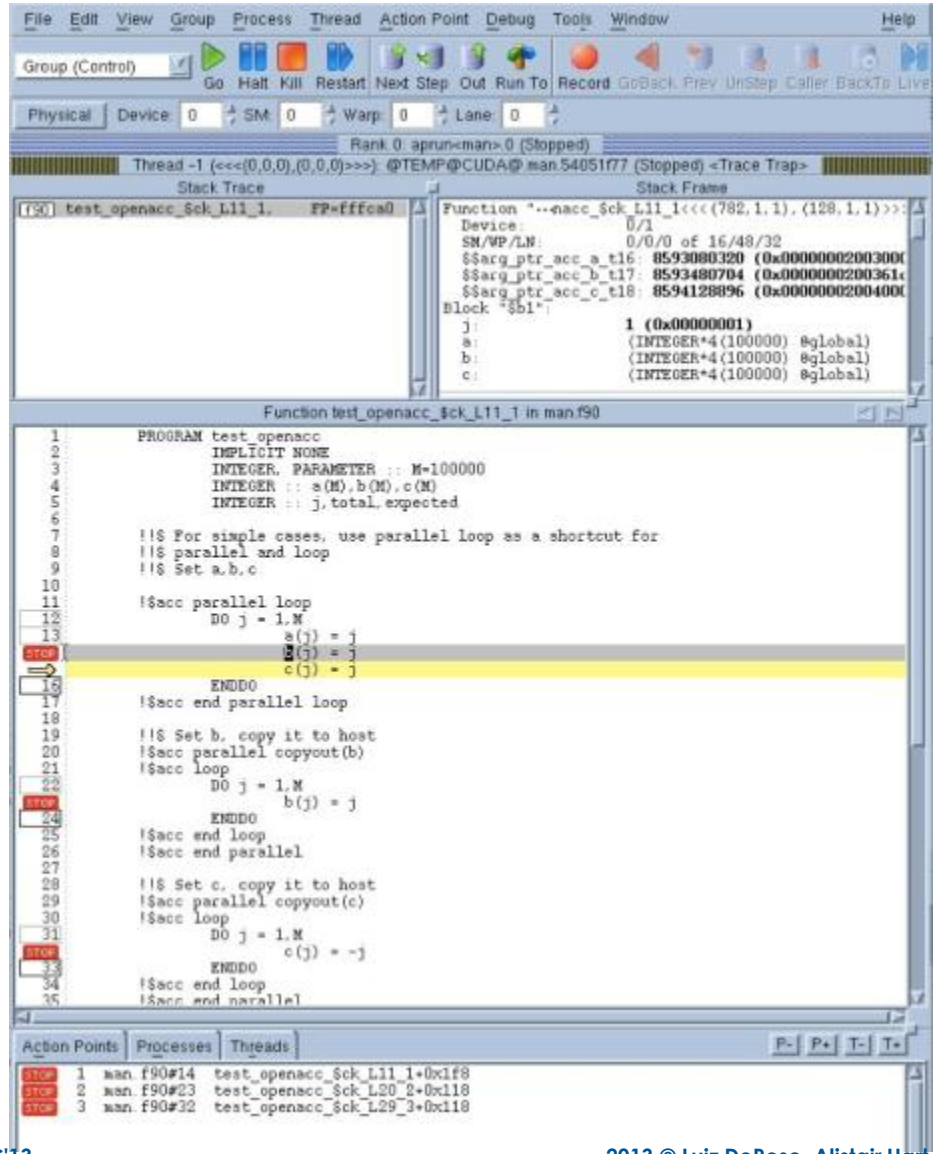
Reference by Logical or Physical attributes



Negative IDs indicate CUDA Threads

Full Thread and Device information in Stack Frame Pane

TotalView for OpenACC



ROGUE WAVE SOFTWARE

OpenACC®

DIRECTIVES FOR ACCELERATORS

- Step host & device
- View variables
- Set breakpoints

Compatibility with Cray CCE 8.x OpenACC

Porting a Parallel Code

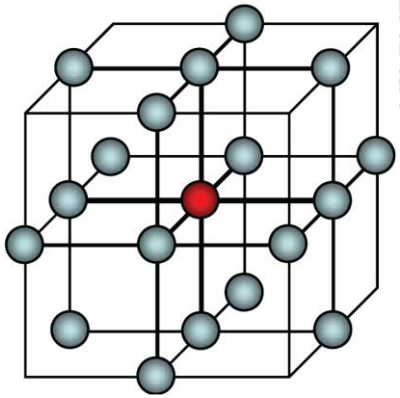
Overview

- **A parallel code is a scalar code with data transfers**
  - We have looked at how to port a scalar code
    - optimising the scalar part procedes in the same way
    - although the local problem size will change when strong scaling
  - Here we look at the parallel version of the same code
- **The new feature is the data transfer between PEs**
  - which also means local data transfers between CPU and GPU
- **This talk looks at the extra things we need to consider**
  - First at a conceptual level
    - the OpenACC part
  - Then some specific points for particular communication models
    - we just discuss MPI here
    - although we also have a version using Fortran coarrays

Contents

- **Himeno benchmark**
  - Structure of the parallel code
- **Packing and transferring halo buffers**
  - async clause
  - wait clause
- **MPI communication**
  - non-blocking sends and receives
  - MPI_WAITANY vs. MPI_WAITALL
  - G2G optimisations

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/2444.htm**
  - Cray also has a Fortran coarray (CAF) version
- **Productivity of OpenACC**
  - ~600 lines of Fortran
  - Fully ported to accelerator using around 30 directive pairs

Overall program structure

- **Like scalar case:**
  - initmt() initialises data
  - jacobi(nn,gosa)
    - does nn iterations
    - stencil update to data
  - called twice:
    - once for calibration
    - once for measurement
- **differences:**
  - initcomm() routine
    - sets up processor grid

```
PROGRAM himeno
   CALL initcomm    ! Set up processor grid
   CALL initmt      ! Initialise local matrices

   cpu0 = gettime() ! Wraps SYSTEM_CLOCK
   CALL jacobi(3,gosa)
   cpu1 = gettime()
   cpu = cpu1 - cpu0

!  nn = INT(ttarget/(cpu/3.0)) ! Fixed runtime
   nn = 1000        ! Hardwired for testing

   cpu0 = gettime()
   CALL jacobi(nn,gosa)
   cpu1 = gettime()
   cpu = cpu1 - cpu0

   xmflops2 = flop*1.0d-6/cpu*nn

   PRINT *,' Loop executed ',nn,' times'
   PRINT *,' Gosa :',gosa
   PRINT *,' MFLOPS:',xmflops2,'  time(s):',cpu
END PROGRAM himeno
```

The distributed jacobi routine

- iteration loop:
  - fixed tripcount
- jacobi stencil:
  - temporary array **wrk2**
  - local residual **wgosa**
- halo exchange
  - between neighbours
  - uses send, receive buffers
- Residual
  - global residual **gosa**
  - **wgosa** summed over PEs
- second kernel:
  - **p** updated from **wrk2**

```
iter_lp: DO loop = 1,nn

  compute stencil: wrk2, wgosa from p

  pack halos from wrk2 into send bufs

  exchange halos with neighbour PEs

  sum wgosa across PEs

  copy back wrk2 into p

  unpack halos into p from recv bufs

ENDDO iter_lp
```

The Jacobi computational kernel (serial)

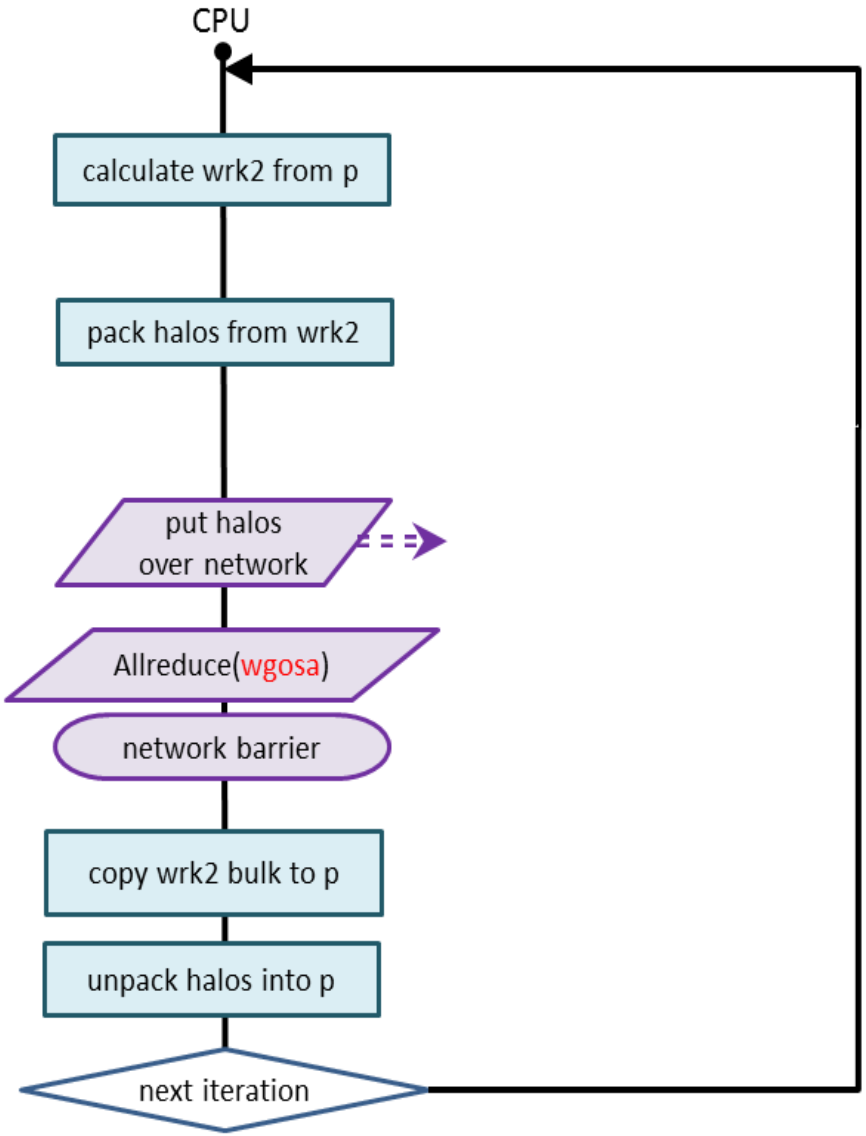
- The stencil is applied to pressure array **p**
- Updated pressure values are saved to temporary array **wrk2**
- Local control value **wgosa** is computed

```
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)

   s = (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.

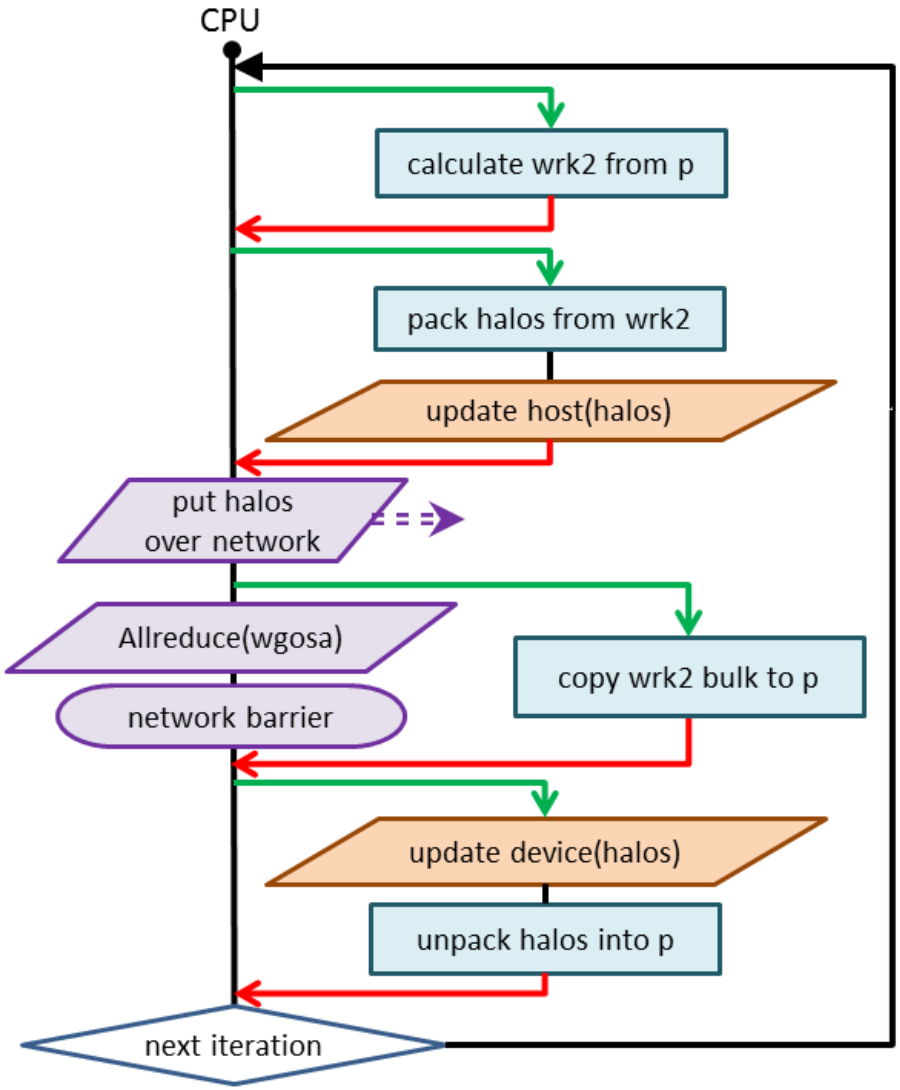
Distributed jacobi routine as a flowchart

- take advantage of overlap
  - just network at this point
- still some freedom:
  - barrier can move relative to
    - Allreduce
    - wrk2→bulk
- Which is best order?
  - depends on:
    - hardware
    - comms library
    - kernel details
    - local problem size
  - Could autotune this
    - separately
    - at runtime



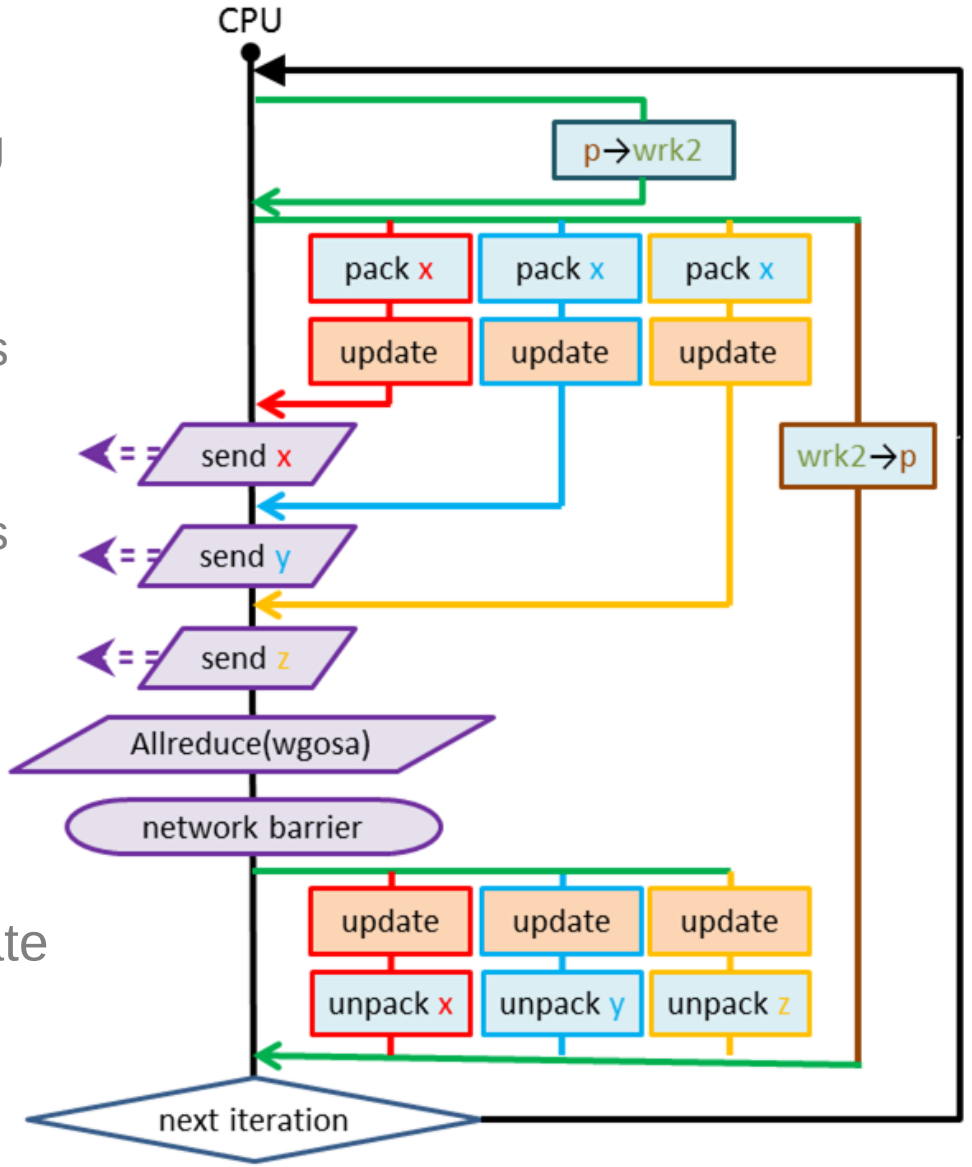
First OpenACC port

- **Loopnest kernels**
  - stencil
  - pack halos: x, y, z
  - wrk2→bulk
  - unpack halos: x, y, z
- **acc updates**
  - 6 buffers each time
- **Some additional overlap**
  - GPU kernels asynchronous
  - wrk2→bulk
    - can overlap with comms



Asynchronicity

- **async clause**
  - allows task overlap on GPU
  - halo pack/update
    - can overlap x, y, z directions
  - halo update/unpack
    - can overlap x, y, z directions
  - can also overlap **`wrk2→p`**
- **Host/GPU sync points**
  - After first kernel
  - After each buffer pack/update
  - At the end of the iteration



Packing and transferring send buffers

- **Use async clause**
  - separate handles for overlap
    - one per direction
      - three in total
      - integers, e.g. 1, 2, 3
  - what about six streams?
    - one each for up and down
    - was not as efficient
      - but this is a tuning option
  - global wait
    - all 3 streams completed

```
!$acc parallel loop async(xstrm)
DO k = 2,kmax-1
  DO j = 2,jmax-1
    sendbuffx_dn(j,k)=wrk2(2,j,k)
    sendbuffx_up(j,k)=wrk2(imax-1,j,k)
  ENDDO
ENDDO
!$acc end parallel loop

!$acc update host &
!$acc    (sendbuffx_dn,sendbuffx_up) &
!$acc   async(xstrm)

! same for y with async(ystrm)
! same for z with async(zstrm)

!$acc wait
<send the 6 buffers>
```

Packing and transferring send buffers (better)

- **The biggest bottleneck**
  - PCIe link serialises:
    - only one buffer moves at a time
  - A better strategy:
    - don't wait for all to finish moving
    - as soon as one direction done
    - send these buffers over network
  - Ordering:
    - x pack/update starts first
    - Can we guarantee x ready first?
    - No, but it is likely
  - No OpenACC "waitany" facility
    - so we go with most likely ordering

```
!$acc parallel loop async(xstrm)
<pack the two x buffers>
!$acc end parallel loop

!$acc update host &
!$acc (sendbuffx_dn,sendbuffx_up) &
!$acc  async(xstrm)

! same for y with async(ystrm)
! same for z with async(zstrm)

!$acc wait(xstrm)
<send the two x buffers>

!$acc wait(ystrm)
<send the two y buffers>

!$acc wait(zstrm)
<send the two z buffers>
```

Transferring and unpacking recv buffers

- **Copying `wrk2` into `p`**
  - very quick kernel
  - but can overlap with network
    - could start this kernel before we sent the data
- **When messages complete**
  - copy and unpack recv buffers
  - three parallel streams again
- **wait**
  - ensures all 4 streams complete
    - 3 update/unpacks plus bulk
  - ready to start next iteration

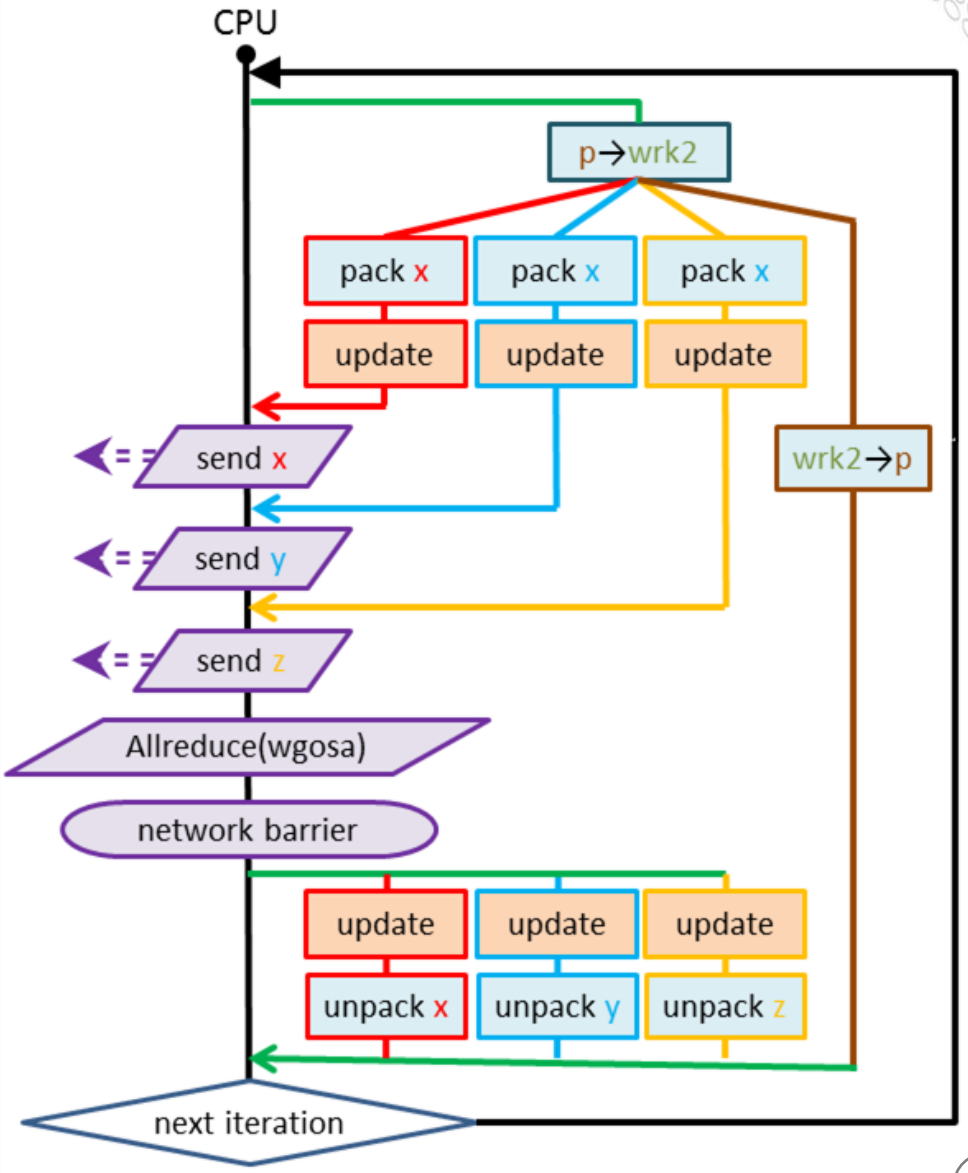
```
!$acc parallel loop async(bstrm)
<copy bulk wrk2 -> p>
!$acc end parallel loop

<network barrier>

!$acc update device &
!$acc  (recvbuffx_dn,recvbuffx_up) &
!$acc  async(xstrm)

!$acc parallel loop async(xstrm)
<unpack the two x buffers>
!$acc end parallel loop

! same for y with async(ystrm)
! same for z with async(zstrm)

!$acc wait

<end iteration loop>
```

Going further with OpenACC v2.0

- **OpenACC v1.0 provides some scope for overlap**
  - async clause
    - allows overlap of separate streams of tasks
    - each stream is a linear sequence of tasks
      - tasks within single stream guaranteed to execute in order
        - but not necessarily immediately one after another;
        - can be delays between
  - wait directive
    - allows host-side synchronisation of one or all streams
- **OpenACC v1.0 does not provide**
  - a way to set up a dependency tree, e.g. to say:
    - "When you have finished these streams of tasks, start this one", or
    - "When you have finished this stream of tasks, start these ones"
- **OpenACC v2.0 does provide this, via the new wait clause**
  - apply to parallel, kernels, update, enter data and exit data directives
  - avoids host-side synchronisation
    - gives greater scope for CPU to do other tasks, uninterrupted

Using the OpenACC v2.0 wait clause

- **Launch 5 operations**
  - stencil kernel
  - 3 buffer packs/updates
    - depends on stencil kernel
  - 1 bulk unpack
    - also depends on stencil
- **wait clause**
  - sorts out dependencies
- **host/device sync points**
  - no longer need after initial, stencil kernel



Using the OpenACC v2.0 wait clause

- In more detail:
- Introduce new stream
  - jstrm for stencil kernel
  - buffer-packing streams wait on this completing
  - acc wait(xstrm) guarantees jstrm has also completed
    - (also waits on ystrm, zstrm)
- bulk copy kernel also waits on jstrm
  - we don't set jstrm=bstrm
  - want to kick-off buffer packs as soon as **wrk2** is ready

```
!$acc parallel loop async(jstrm)
<stencil p -> wrk2>

!$acc parallel loop &
!$acc   async(xstrm) wait(jstrm)
<pack x buffers from wrk2>

!$acc update host &
!$acc   (sendbuffx_dn,sendbuffx_up) &
!$acc   async(xstrm)

! same for y with async(ystrm)
! same for z with async(zstrm)

!$acc parallel loop &
!$acc   async(bstrm) wait(jstrm)
<copy bulk wrk2 -> p>

!$acc wait(xstrm)
<send the two x buffers>

! same for y with async(ystrm)
! same for z with async(zstrm)
```

Data region considerations

- **Twelve extra buffers needed**
  - 3 physical dimensions: x, y, z
    - 2 directions of transfer: up, down
      - 2 separate buffers: send, receive
- **These buffers need to be added to the data region(s)**
  - if just in jacobi() data region
    - they should be create
      - allocated each time jacobi() is called, no data transfer (except explicit updates)
  - or in outer data region
    - create in main data region
    - present in jacobi() data region
    - need to be declared globally
  - *advice:*
    - *if jacobi() routine is called a lot, often better to allocate once in parent*

Communications models

- **Two domain decompositions strategies**
  - Both message passing:
    - MPI
    - Fortran coarrays (CAF)
- **A lot of similarities**
  - a few subtle differences

MPI

- **Use nonblocking MPI calls**
  - MPI_IRECV
    - post these receives early; first thing in iteration loop
  - MPI_ISEND
    - call these in 3 sets (x, y, z) of two (up, down)
      - after each of async streams xstrm, ystrm, zstrm separately completes
  - network barrier:
    - MPI_WAITALL(12,send_recv_handles(:))
    - ensures all buffers arrive (and send buffers can be reused)
  - Then kick-off the three streams copying recv buffers to accelerator and unpacking

Better host-side MPI

- **PCIe transfer of three sets of recv buffers is a bottleneck**
  - buffer transfers will serialise, one at once
  - better to start transfer of messages to GPU as soon as each arrives
    - don't know in which order the 6 message will arrive
- **Rather than MPI_WAITALL(12,send_recv_handles(:))**
  - Loop over MPI_WAITANY(6,recv_handles(:))
    - As each arrives, start separate async stream comprising:
      - update, then unpack kernel
    - 6 different streams being used here
- **After all MPI recvs completed:**
  - MPI_WAITALL(6, send_handles(:))
    - so we the send buffers are ready for reuse in the next iteration
  - acc wait ensures 7 streams have completed
    - 6 copy/unpacks, plus the bulk stream
- **Could also use MPI3 nonblocking collective for gosa**
  - would MPI_WAIT on this handle just before end of iteration loop

Improved G2G MPI

- **Latest Cray MPICH library offers "G2G" feature**
  - Can call MPI on CPU, but passing GPU buffer to library
  - Data moves between GPU and network seamlessly
    - no need for update directives
    - no need for host-side synchronisation between update and send/recv
      - when MPI completes, know data has moved
      - fewer sync points allows more flexible overlap of CPU and GPU operations
    - improved end-to-end communication bandwidth
- **host_data directive used to expose device data pointers**
  - Put host_data regions around MPI calls
- **Code can still be compiled for the CPU**

Using the G2G MPI

- **Receives**
  - posted with GPU buffers
- **updates removed**
  - just pack buffers on GPU
- **Sends**
  - also use GPU addresses
- **MPI_WAITALL**
  - data now moved to recv buffers on remote GPUs
  - no extra GPU sync needed
  - no need for MPI_WAITANY any more
    - can use 3 unpack streams

```
!$acc host_data use_device &
        (recvbuffx_dn,recvbuffx_up)
call MPI_IRECV(recvbuffx_dn,...)
call MPI_IRECV(recvbuffx_up,...)
!$acc end host_data
! same for y,z

<stencil p -> wrk2> async(jstrm)

<pack x buffs> async(xstrm) wait(jstrm)
! same for y,z

!$acc wait(xstrm)
!$acc host_data use_device &
        (sendbuffx_dn,sendbuffx_up)
call MPI_ISEND(sendbuffx_dn,...)
call MPI_ISEND(sendbuffx_up,...)
!$acc end host_data
! same for y,z

call MPI_WAITALL(12,...)

<unpack x buffs> async(xstrm)
! same for y,z

!$acc wait
```

Summary

- **A parallel code is just like a scalar code, but with data transfers**
  - These data transfers add additional complications
  - Provide a lot of scope for overlap:
    - overlapping GPU kernels/transfers
    - overlapping compute and network transfers
  - A lot of scope implies a lot of algorithmic choices
    - Which is best will depend on the code, the problem, the parallel decomposition and the hardware
    - An autotuning approach may help
- **General advice:**
  - Profile the production code frequently and optimise the slowest bit
    - Could be kernel performance or network transfers
    - The balance will shift as you continue optimising

Summary

- **Some things to look for**
  - look for opportunities to overlap GPU tasks with the async clause
    - e.g. packing buffers in different directions
  - look for opportunities to exploit the wait clause
    - going beyond simple streams of tasks to a dependency tree
  - use asynchronous MPI
    - if you are not already doing so
    - extra PCIe transfer time gives more scope for network overlap
  - try MPI_WAITANY rather than MPI_WAITALL
    - to exploit this potential for extra overlap
  - or use G2G MPI to remove host-side synchronisation points
    - and improve data transfer rates between accelerators on different nodes
- **The same principles apply for other message-passing programming models**
  - CAF, SHMEM etc.

OpenACC 3.0 and OpenMP 4.0+
A road map

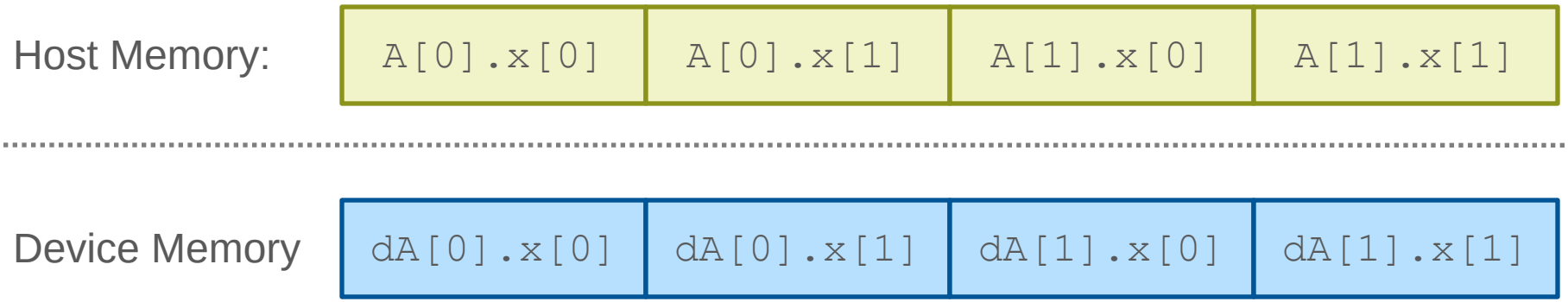
OpenACC 2.1 and beyond

- Full support for C and C++ structs and struct members, including pointer members.
- Full support for Fortran derived types and derived type members, including allocatable and pointer members.
- Defined support with multiple host threads.
- Optionally removing the synchronization or barrier at the end of vector and worker loops.
- Allowing an **if** clause after a **device_type** clause.
- A **default(none)** clause for the loop directive.
- A **shared** clause (or something similar) for the loop directive.
- A standard interface for a profiler or trace or other runtime data collection tool.
- Better support for multiple devices from a single thread, whether of the same type or of different types.

Flat object model

- **OpenACC supports a "flat" object model**
  - Primitive types
  - Composite types without allocatable/pointer members

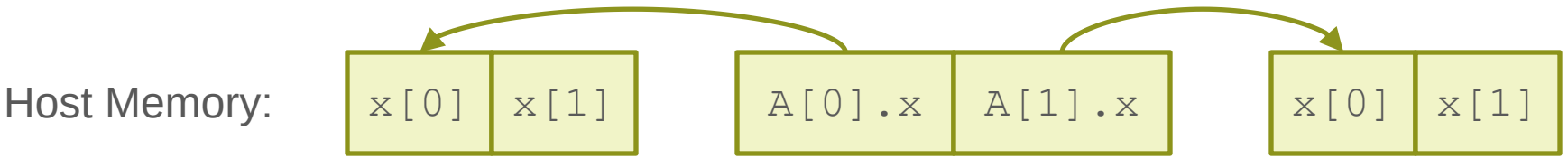
```
struct {
  int x[2]; // static size 2
} *A;       // dynamic size 2
#pragma acc data copy(A[0:2])
```



Challenges with pointer indirection

- Non-contiguous transfers
- Pointer translation

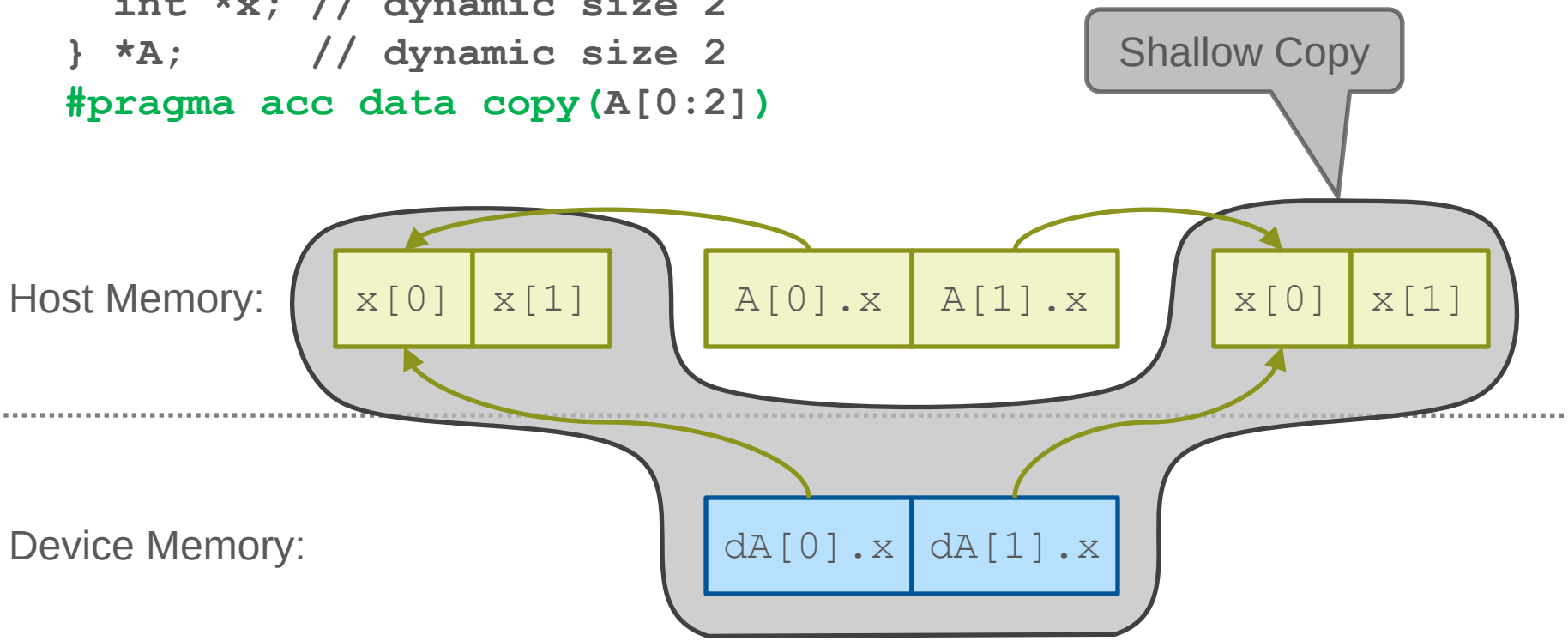
```
struct {
  int *x; // dynamic size 2
} *A;     // dynamic size 2
#pragma acc data copy(A[0:2])
```



Challenges with pointer indirection

- Non-contiguous transfers
- Pointer translation

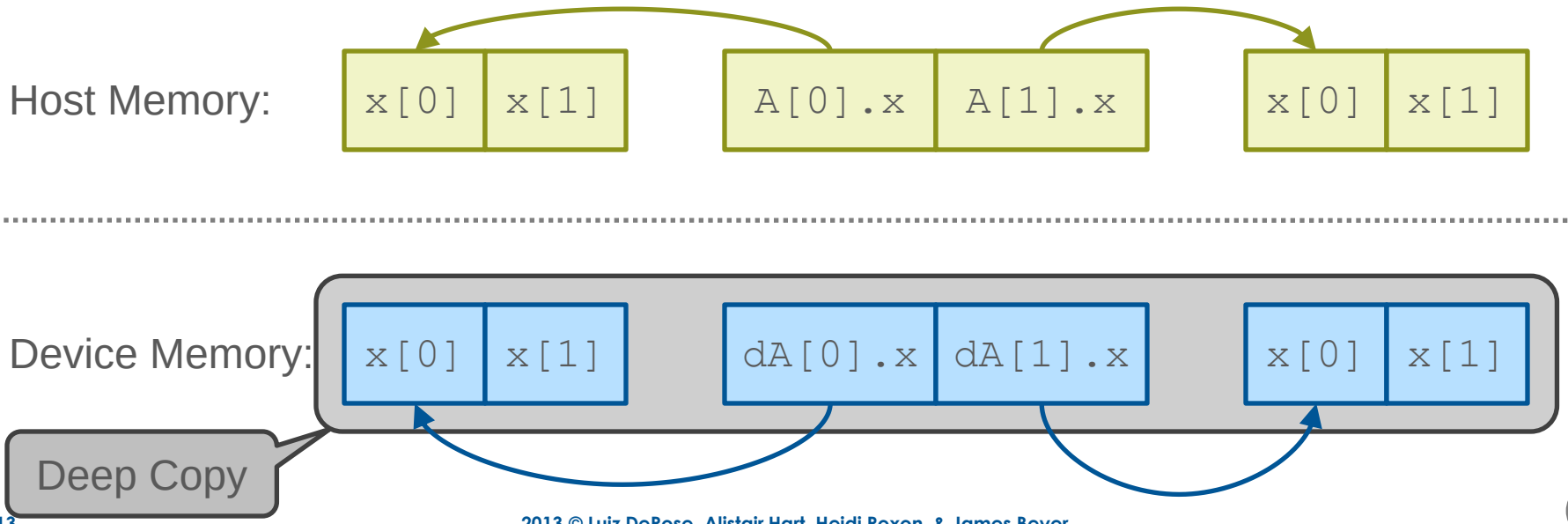
```
struct {
  int *x; // dynamic size 2
} *A;     // dynamic size 2
#pragma acc data copy(A[0:2])
```



Challenges with pointer indirection

- Non-contiguous transfers
- Pointer translation

```
struct {
  int *x; // dynamic size 2
} *A;     // dynamic size 2
#pragma acc data copy(A[0:2])
```



Possible deep-copy solutions

- **Re-write application**
  - Use “flat” objects
- **Manual deep copy**
  - Issue multiple transfers
  - Translate pointers
- **Compiler-assisted deep copy**
  - Automatic for fortran
    - -hacc_models=deep_copy
    - Dope vectors are self describing
  - OpenACC extensions for C/C++
    - Pointers require explicit shapes

Appropriate for CUDA

Appropriate for OpenACC

Manual deep-copy

```
struct A_t
  int n;
  int *x;       // dynamic size n
};
...
struct A_t *A; // dynamic size 2
/* shallow copyin A[0:2] to device_A[0:2] */
struct A_t *dA = acc_copyin( A, 2*sizeof(struct A_t) );
     int i = 0 ; i < 2 ; i++) {
  /* shallow copyin A[i].x[0:A[i].n] to "orphaned" object */
  int *dx = acc_copyin( A[i].x, A[i].n*sizeof(int) );
  /* fix acc pointer device_A[i].x */
  cray_acc_memcpy_to_device( &dA[i].x, &dx, sizeof(int*);
}
```

- **Currently works for C/C++**
  - Fortran programmers have to know the tricks
- **Portable in OpenACC 2.0, but not usually practical**

Proposed “member shape” directives

```
struct A_t {
  int n;
  int  x;       // dynamic size n
#pragma acc declare shape(x[0:n])
};
...
struct A_t *A; // dynamic size 2
...
/* deep copy */
#pragma acc data copy(A[0:2])
```

- Each object must shape it’s own pointers
- Member pointers must be contiguous
- No polymorphic types (types must be known statically)
- Pointer association may not change on accelerator (including allocation/deallocation)
- Member pointers may not alias (no cyclic data structures)
- Assignment operators, copy constructors, constructors or destructors are not invoked

Member-shape directive examples

```
extern int size_z();
int size_y;
struct Foo
{
  double* x;
  double* y;
  double* z;
  int     size_x;
  // deep copy x, y, and z
  #pragma acc declare shape(x[0:size_x], y[1:size_y-1], z[0:size_z()])

type Foo
    real,allocatable :: x(:)
    real,pointer     :: y(:)
    !$acc declare shape(x)   ! deep copy x
    !$acc declare unshape(y) ! do not deep copy y
end type Foo
```

OpenMP 4.0

OpenMP accelerator directives

- **Announced 15yrs ago**
- **Works with Fortran, C, C++**
- **An established open standard is the most attractive**
  - portability; multiple compilers for debugging; permanence
- **Last version 3.1 (July 2011)**
  - A common directive programming model for shared memory systems
  - 354 pages
- **Latest version 4.0 (July 2013)**
  - A common programming model for wide range of accelerators
  - 320 pages
  - 140 pages moved to an examples document!

- **OpenACC will continue to be supported**
  - Developers can transition to OpenMP if they wish
  - Converting OpenACC to OpenMP will be straightforward
    - More on this later

OpenMP 4.0 “device” additions

- **Target data**
  - Place objects on the device
- **Target**
  - Move execution to a device
- **Target update**
  - Update objects on the device or host
- **Declare target**
  - Place objects on the device
  - Place subroutines/functions on the device
- **Teams**
  - Start multiple contention groups
  - This gains access to the ThreadBlocks
- **Distribute**
  - Similar to the OpenACC loop construct, binds to teams construct
  - Distribute simd
  - Distribute parallel loop
  - Distirbute parallel loop simd
- **Array sections**

OpenACC compared to OpenMP

OpenACC	OpenMP
Parallel (offload)	**Target**
Parallel (multiple “threads”)	**Team/Parallel**
Kernels	
Data	**Target Data**
Loop	**Distribute/Do/for**
Host data	
Cache	
Update	**Target Update**
Wait	
Declare	**Declare target**

OpenACC compared to OpenMP continued

OpenACC	OpenMP
enter data	
exit data	
data api	
routine	declare target
async wait	
parallel in parallel	Parallel in parallel or team
Tile	
Device_type	

OpenACC compared to OpenMP continued

OpenACC	OpenMP
atomic	Atomic
	Critical sections
	Master
	Single
	Tasks
	barrier
	get_thread_num
	get_num_threads
	…

OpenMP async

- **Target does NOT take an async clause!**
  - Does this mean no async capabilities?
- **OpenMP already has async capabilities -- Tasks**
  - !$omp task
  - #pagma omp task
  - depend(dependence-type: list)
- **Is this the best solution?**

Transition from OpenACC to OpenMP

- **OpenACC 1.0 to OpenMP 4.0 is straight forward**
- **OpenACC 2.0 to OpenMP 4.0 has issues**
  - Unstructured data lifetimes
  - Tile
- **OpenMP 4.1 and 5.0 should close many of the existing gaps**
- **Differences are significant enough that OpenACC may never fold back into OpenMP**
  - OpenACC aims for portable performance
  - OpenMP aims for programmability

Sources of further information

- **Standards web pages:**
  - OpenACC.org
  - OpenMP.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
  - OpenMP forums: openmp.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)

!$acc end data