An Introduction to OpenACC

James Beyer PhD

Timetable

Monday 6th May 2013

8:30	Lecture 1:	Introduction to the Cray XK7	(15)
8:45	Lecture 2:	OpenACC organization (Duncan Poole)	(15)
9:00	Lecture 3:	The OpenACC programming model	(30)
9:30	Lecture 4:	Porting a simple example to OpenACC	(30)
10:00	*break*		*(30)*
10:30	Lecture 5:	Advanced OpenACC	(40)
11:10	Lecture 6:	Using CCE with OpenACC	(25)
11:35	Lecture 7:	OpenACC 2.0 and OpenMP 4.0	(25)
12:00	*close*		

Contents

- **The aims of this course:**
  - To motivate why directive-based programming of GPUs is useful
  - To introduce you to the OpenACC programming model
  - To give you some experience seeing OpenACC directives in a code
- **The idea is to prepare you for future tutorials and initial porting efforts**

Inside the Cray XK7
and the Nvidia Kepler K20X GPU

Contents of this talk

- **An overview of the Cray XK7**
  - The hardware
  - Why GPUs are interesting for Exascale research
  - Programming models for GPUs
- **A quick GPU refresher**
  - the hardware
  - how codes execute on the hardware and what this means to the programmer
- **Things to consider before starting an OpenACC port**

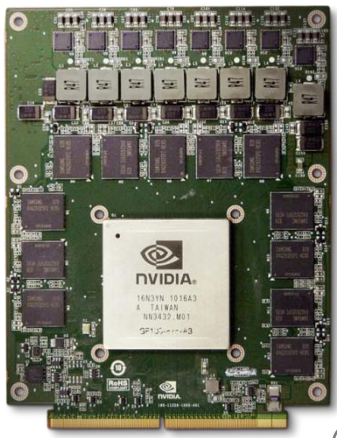
"Accelerating the Way to Better Science"

Cray XK(6|7) supercomputer

- **Node architecture:**
  - One AMD Series 6200 Interlagos CPU (16 cores)
  - One Nvidia GPU
    - XK6 Fermi+
      - 512 cores, 665 GFlop/s DP, 6GB memory
    - XK7 Kepler
      - 2496 cores, 1.17 TFlop/s DP, 5GB memory
      - 2688 cores, 1.31 TFlop/s DP, 6GB memory

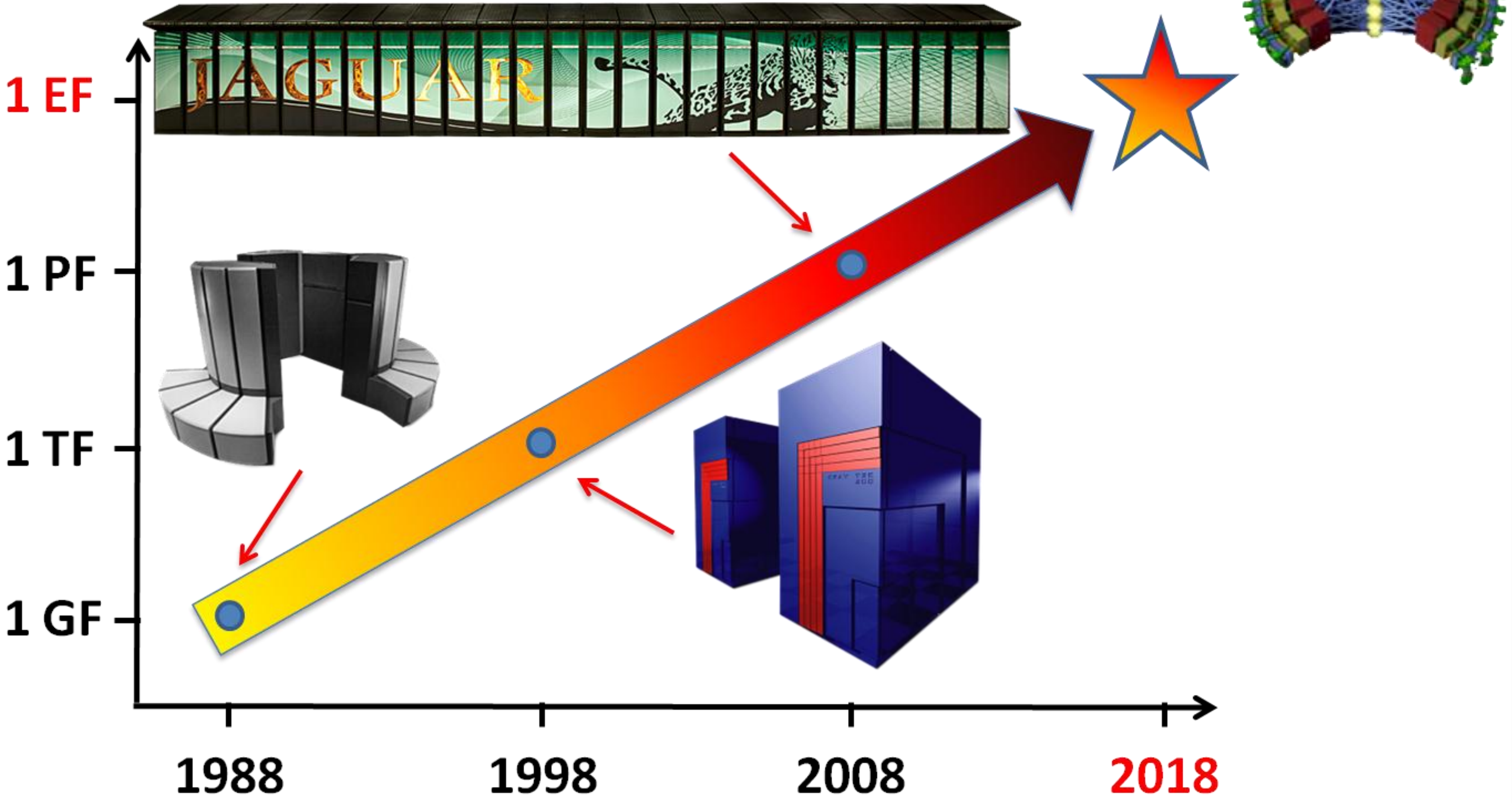
- **Cray Gemini interconnect**
    - shared between two nodes
    - high bandwidth/low latency scalability

- **Fully integrated/optimized/supported**
    - Tight integration of GPU and NIC drivers



The Exascale is coming...

- Sustained performance milestones every 10 years...
  - 1000x the performance with 100x the PEs



(and they're all Crays)

Exascale, but not exawatts

- **Power is a big consideration in an exascale architecture**
  - Jaguar XT (ORNL) draws 6MW to deliver 1PF
  - The US DoE wants 1EF, but using only 20MW...
- **A hybrid system is one way to reach this, e.g.**
  - 10^5 nodes (up from 10^4 for Jaguar)
  - 10^4 FPUs/node (up from 10 for Jaguar)
    - some full-featured cores for serial work
    - a lot more cutdown cores for parallel work
  - Instruction level parallelism will be needed
    - continues the SIMD trend SSE → AVX → ...
- **This looks a lot like the current GPU accelerator model**
  - manycore architecture, split into SIMT threadblocks
  - Complicated memory space/hierarchy (internal and PCIe)
- **And this looks a lot like the old days**
  - welcome back to vectorization, we kept the compiler ready for you

Accelerator programming

- **Why do we need a new GPU programming model?**
- **Aren't there enough ways already?**
  - CUDA (incl. NVIDIA CUDA-C & PGI CUDA-Fortran)
  - OpenCL
  - Stream
  - hiCUDA ...
- **All are quite low-level and closely coupled to the GPU**
  - User needs to rewrite kernels in specialist language:
    - Hard to write and debug
    - Hard to optimise for specific GPU
    - Hard to port to new accelerator
  - Multiple versions of kernels in codebase
    - Hard to add new functionality

Directive-based programming

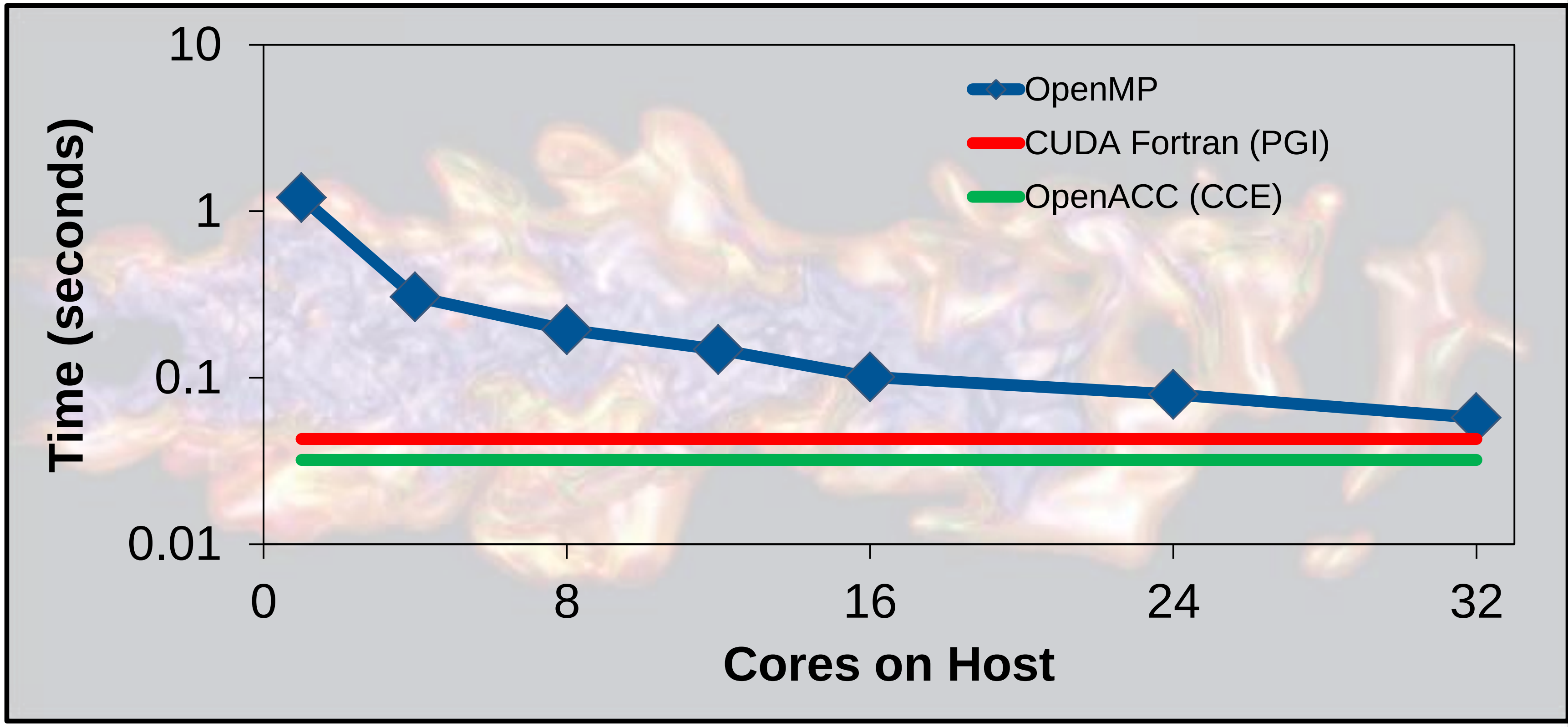
Directives provide a high-level alternative

- + **Based on original source code (e.g. Fortran, C, C++)**
  - + Easier to maintain/port/extend code
  - + Users with (for instance) OpenMP experience find it a familiar programming model
  - + Compiler handles repetitive boilerplate code (cudaMalloc, cudaMemcpy...)
  - + Compiler handles default scheduling; user can step in with clauses where needed

- – **Possible performance sacrifice**
  - – Important to quantify this
  - – Can then tune the compiler
  - – Small performance sacrifice is an acceptable trade-off for portability and productivity
    - – After all, who handcodes in assembly for CPUs these days?

Performance compared to CUDA

- Is there a performance gap relative to explicit low-level programming model? Typically 10-15%, sometimes none.
- Is the performance gap acceptable? Yes.
  - e.g. S3D comp_heat kernel (ORNL application readiness):



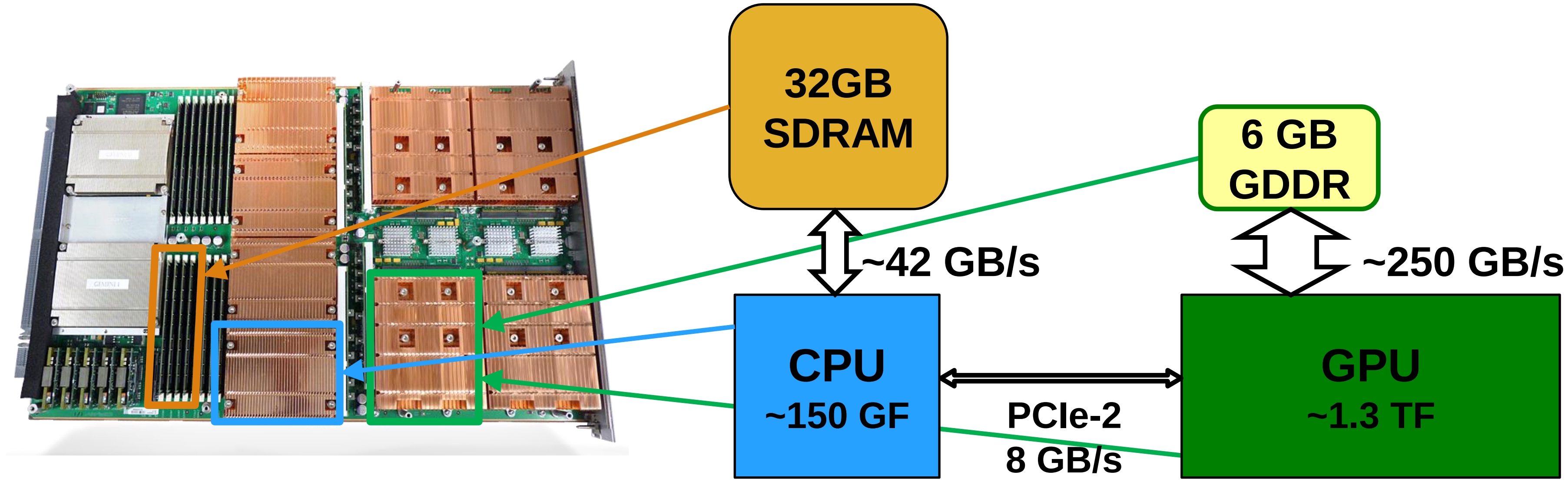
Structure of this course

- **Aims to lead you through the entire development process**
  - What is OpenACC?
  - How do I use it in a simple code?
  - Performance tuning and advanced topics
- **It will assume you know**
  - A little bit about GPU architecture and programming
    - SMs, threadblocks, warps, coalescing
    - a quick refresher follows
- **It will help if you know**
  - The basic idea behind OpenMP programming
    - but this is not essential

A quick GPU refresher

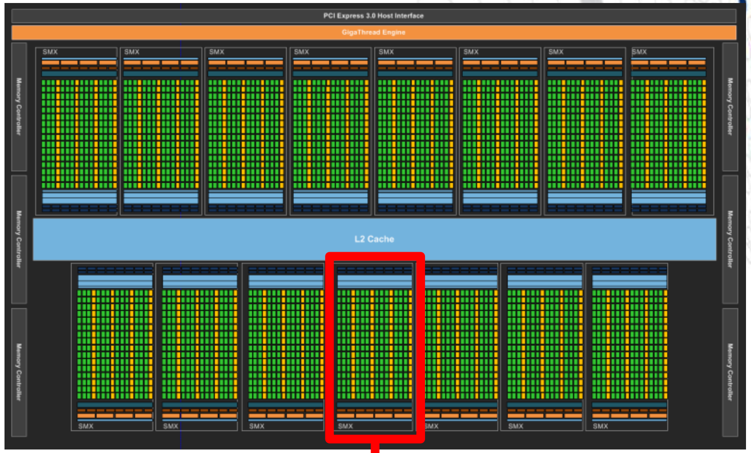
How fast are current GPUs?

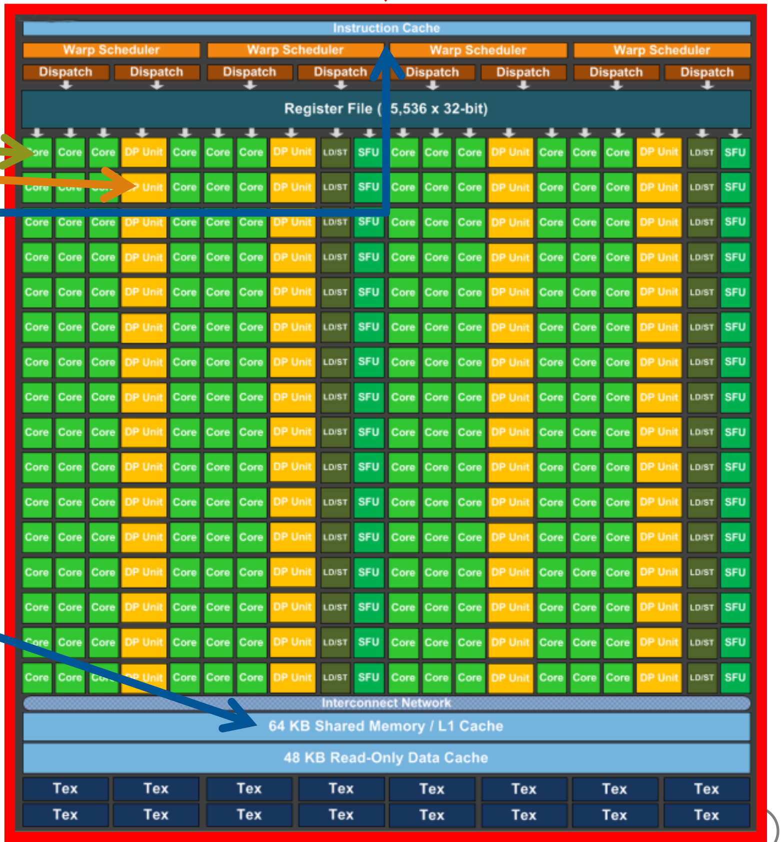
- **Beware the hype: "I got 1000x speed-up on a GPU"**
- **What should you expect?**
  - Cray XK7:
    - Flop/s: GPU ~9x faster than single, whole CPU (16 cores)
    - Memory bandwidth: GPU ~6x faster than CPU
  - These ratios are going to be similar in other systems
- **Plus, it is harder to reach peak performance on a GPU**
  - Your code needs to fit the architecture
  - You also need to factor in data transfers between CPU and GPU



Nvidia K20X Kepler architecture

- **Global architecture**
  - a lot of compute cores
    - 2688 SP plus 896 DP; ratio 3:1
  - divided into 14 Streaming Multiprocessors
  - these operate independently
- **SMX architecture**
  - many cores
    - 192 SP
    - 64 DP
  - shared instruction stream; same ops
    - lockstep, SIMT execution of same ops
    - SMX acts like vector processor
- **Memory hierarchy**
  - each core has private registers
    - fixed register file size
  - cores in an SM share a fast memory
    - 64KB, split between:
      - L1 cache and user-managed
  - all cores share large global memory
    - 6GB; also some specialist memory




Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

Issues around GPUs and OpenACC

- **Program Execution on a GPU**
  - Kernels are launched by CPU to execute on GPU
  - The GPU runtime schedules Kernels on hardware
  - Kernel launch is asynchronous
- **What CUDA doesn't tell you (upfront)**
  - Threads are not created equal
    - warps
  - Memory accesses done at the warp level
  - Compiler looks at GPU as a SMP vector processor
- **What does this mean to programmers**
  - Need a lot of parallel tasks
  - Loops must vectorize
  - Data transfers are expensive
  - Synchronization is not possible at ThreadBlock level
- **With Auto-vectorization do we need directives?**
  - Location location location
- **Risk Factors**
  - Will there be machines to run my code?
  - Will OpenACC continue?
  - Will OpenACC be superseded?

OpenACC Organization

Duncan Poole

Timetable

Monday 6th May 2013

8:30	Lecture 1:	Introduction to the Cray XK7	(15)
8:45	Lecture 2:	OpenACC organization (Duncan Poole)	(15)
9:00	Lecture 3:	The OpenACC programming model	(30)
9:30	Lecture 4:	Porting a simple example to OpenACC	(30)
10:00	*break*		*(30)*
10:30	Lecture 5:	Advanced OpenACC	(40)
11:10	Lecture 6:	Using CCE with OpenACC	(25)
11:35	Lecture 7:	OpenACC 2.0 and OpenMP 4.0	(25)
12:00	*close*		

OpenACC®
DIRECTIVES FOR ACCELERATORS

- **A common directive programming model for today's GPUs**
  - Announced at SC11 conference
  - Offers portability between compilers
    - Drawn up by: NVIDIA, Cray, PGI, CAPS
    - Multiple compilers offer:
      - portability, debugging, permanence
  - Works for Fortran, C, C++
    - Standard available at openacc.org
    - Initially implementations targeted at NVIDIA GPUs
- **Current version: 1.0 (November 2011)**
  - v2.0 expected in 1H 2013
- **Compiler support: all now complete**
  - Cray CCE: complete in 8.1 release
  - PGI Accelerator: version 12.6 onwards
  - CAPS: Full support in v1.3
  - (accULL: research compiler, C only)

The OpenACC™ API
QUICK REFERENCE GUIDE

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.

Most OpenACC directives apply to the immediately following structured block or loop; a structured block is a single statement or a compound statement (C or C++) or a sequence of statements (Fortran) with a single entry point at the top and a single exit at the bottom.

Version 1.0, November 2011

All four companies intend to work within OpenMP organization to merge OpenACC and create a common specification that extends OpenMP to support accelerators.

© 2011 OpenACC-standard.org all rights reserved.

CAPS CRAY THE SUPERCOMPUTER COMPANY NVIDIA. The Portland Group

The OpenACC programming model

James Beyer

Timetable

Monday 6th May 2013

8:30	Lecture 1:	Introduction to the Cray XK7	(15)
8:45	Lecture 2:	OpenACC organization (Duncan Poole)	(15)
9:00	Lecture 3:	The OpenACC programming model	(30)
9:30	Lecture 4:	Porting a simple example to OpenACC	(30)
10:00	*break*		*(30)*
10:30	Lecture 5:	Advanced OpenACC	(40)
11:10	Lecture 6:	Using CCE with OpenACC	(25)
11:35	Lecture 7:	OpenACC 2.0 and OpenMP 4.0	(25)
12:00	*close*		

Contents

- **OpenACC programming model**
- **What does OpenACC looks like?**
- **How are OpenACC directives used?**
  - Basic directives
    - 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 programming model

- **Host-directed execution with attached GPU**
  - Main program executes on “host” (i.e. CPU)
  - Directs execution on device (i.e. GPU)
    - Memory allocation and transfers
    - Kernel execution
    - Synchronization
- **Memory spaces on the host and device distinct**
  - Different locations, 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 synchronization possible between outermost parallel level
- **User responsible for**
  - Specifying code to run on device
  - Specifying parallelism
  - Specifying data allocation/movement that spans single kernels

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 !$acc end *
      - Directives can be capitalised
  - Continuation to extra lines allowed
    - C/C++: \ (at end of line to be continued)
    - Fortran:
      - Fixed form: c$acc& or !$acc& on continuation line
      - Free form: & at end of line to be continued
        - continuation lines can start with either !$acc or !$acc&

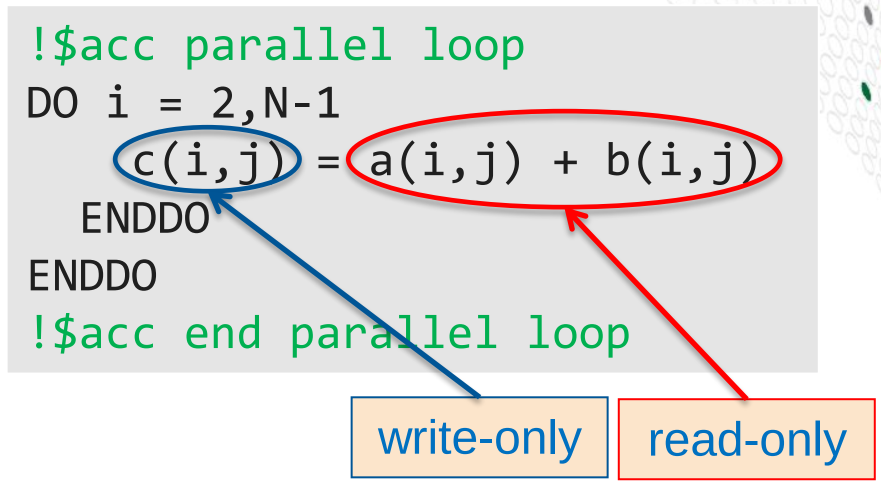
```
// C/C++ example
#pragma acc *
{structured block}
```

```
! 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**
  - E.g.
    - around calls to OpenACC runtime API functions
    - where you need to recode for OpenACC, e.g. for performance reasons
      - try to minimize this; usually better OpenACC code is better CPU code
- **CPP macro defined to allow conditional compilation**
  - _OPENACC == yyyymm (currently 201111)

A first example



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

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
  - 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 (explicitly use GPU shared memory for reused data)
    - automatic caching (e.g. NVIDIA Fermi, Kepler) important
  - Tune default behavior with optional clauses on directives

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 synchronization 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
    - Fermi: Compiler will inline (maybe need `-Oipafrom` or program library)
- **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 call-tree structure of program can be preserved**

Data clauses

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

More data clauses

- present, present_or_copy*, present_or_create
  - pcopy*, pcreate for short
  - Checks if data is already on the device
    - if it is, it uses that version
      - no data copying will be carried out for that data
    - if not, it does the prescribed data copying
  - Advice: only use present_or_* if you really have to
    - "not present" runtime errors are a useful development tool for most codes

- In both cases, the data is processed on the GPU
- Advanced topic: what if I want to call routine either:
  - with data on the GPU, to be processed on the GPU, or...
  - with data on the CPU, to be processed on the CPU?
- Either:
  - Explicitly call one of two versions of the routine, one with OpenACC, or...
  - Use the Cray OpenACC runtime to check if data present and branch code

And take a breath...

- **You now know everything you need to start accelerating**
- **So what do we do for the rest of the lecture?**
  - Not all codes are simple
  - OpenACC has a lot more functionality to cover
  - And we want to be able to tune the performance

Clauses for !$acc parallel loop

- **Tuning clauses:**
  - !$acc loop [gang] [worker] [vector]
    - Targets specific loop (or loops with collapse) at specific level of hardware
      - gang ↔ CUDA threadblock (scheduled on a single SM)
      - worker ↔ CUDA warp of 32 threads (scheduled on vector unit)
      - vector↔ CUDA threads in warp executing in SIMT lockstep
    - You can specify more than one
      - !$acc loop gang worker vector schedules loop iteration over all hardware
    - We'll discuss loop scheduling in much more detail later

More clauses for !$acc parallel loop

- **More tuning clauses:**
- **num_gangs, num_workers, vector_length**
  - Tunes the amount of parallelism used (threadblocks, threads/block...)
  - To set the number of threads per block (fixed at compile time for CCE)
    - vector_length(NTHREADS) *or* num_workers(NTHREADS/32)
    - NTHREADS must be one of: 1, 64, 128 (default), 256, 512, 1024
    - NTHREADS > 32 automatically decomposed into warps of length 32

  - Don't need to specify number of threadblocks (unless you want to)

  - Handy tip: To debug a kernel by running on a single GPU thread, use:
    - !$acc parallel [loop] gang vector num_gangs(1) vector_length(1)
    - Useful for checking race conditions in parallelised loopnests (but very slow)

More OpenACC directives

- **Other !$acc parallel loop clauses:**
  - seq: loop executed sequentially
  - independent: compiler hint, if it isn't partitioning (parallelising) a loop
  - if(logical)
    - Executes on GPU if .TRUE. at runtime, otherwise on CPU
  - reduction: as in OpenMP
  - cache: specified data held in software-managed data cache
    - e.g. explicit blocking to shared memory on NVIDIA GPUs
- **CCE-specific tuning:**
  - can also use !dir$ directives to adjust loop scheduling
    - e.g. concurrent, blockable
  - see man intro_directives (with PrgEnv-cray loaded) for details

More OpenACC directives

- **!$acc update [host|device]**
  - Copy specified arrays (slices) within data region
  - Useful if you only need to send a small subset of data to/from GPU
    - e.g. halo exchange for domain-decomposed parallel code
    - or sending a few array elements to the CPU for printing/debugging
  - Remember slicing syntax differs between Fortran and C/C++
  - The contiguous array sections perform better
- **!$acc declare**
  - Makes a variable resident in accelerator memory
    - persists for the duration of the implicit data region
- **Other directives**
  - We'll cover these in detail later:
    - !$acc cache
    - async clause and !$acc wait
    - !$acc host_data

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

parallel loop vs. parallel and loop

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

parallel loop vs. parallel and loop

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

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

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

- **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!

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
  - OpenACC specific
    - will need pre-processing away for CPU execution
    - #ifdef _OPENACC

- **CCE offers an extended runtime API**
  - set of library calls, names starting with cray_acc_
    - will need pre-processing 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.

Sources of further information

- **OpenACC standard web page:**
  - OpenACC.org
    - documents: full standard and quick reference guide PDFs
    - links to other documents, tutorials etc.
- **Discussion lists:**
  - Cray users: openacc-users@cray.com
    - automatic subscription if you have a raven account
  - OpenACC forum: openacc.org/forum
- **CCE man pages (with PrgEnv-cray loaded):**
  - programming model and Cray extensions: intro_openacc
  - examples of use: openacc.examples
  - also compiler-specific man pages: crayftn, craycc, crayCC
- **CrayPAT man pages (with perftools loaded):**
  - intro_craypat, pat_build, pat_report
    - also command: pat_help
  - accpc (for accelerator performance counters)

Porting a simple example to OpenACC: the scalar Himeno code

James Beyer

Timetable

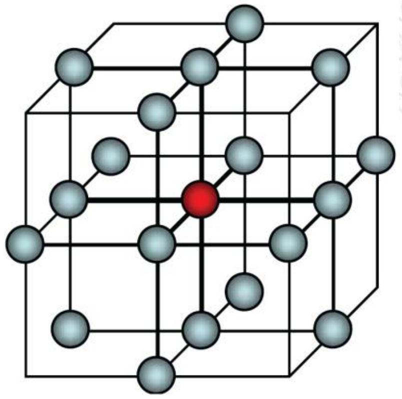
Monday 6th May 2013

8:30	Lecture 1:	Introduction to the Cray XK7	(15)
8:45	Lecture 2:	OpenACC organization (Duncan Poole)	(15)
9:00	Lecture 3:	The OpenACC programming model	(30)
9:30	Lecture 4:	Porting a simple example to OpenACC	(30)
10:00	*break*		*(30)*
10:30	Lecture 5:	Advanced OpenACC	(40)
11:10	Lecture 6:	Using CCE with OpenACC	(25)
11:35	Lecture 7:	OpenACC 2.0 and OpenMP 4.0	(25)
12:00	*close*		

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

The Himeno Benchmark



- **3D Poisson equation solver**
  - Iterative loop evaluating 19-point stencil
  - Memory intensive, memory bandwidth bound
- **Fortran and C implementations available from http://accc.riken.jp/2444.htm**
- **We look at the scalar version for simplicity**
- **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 GPU vs. single CPU core)

- **There are two key things to understand about the code:**
  - 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 a GPU?
        - What are the min/average/max tripcounts?
    - Minimising data movements will probably require eventual acceleration of many more (and possibly all) loopnests, but we have to start somewhere

- **Answering these questions for a large application is hard**
  - There are tools to help (we will discuss some of them later in the course)
  - With a simple code, we can do all of this just by code inspection

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

- **Remember to verify correctness along the way.**
- **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, and each call is timed:
      - Each call is timed internally by the code
      - The first call does a small fixed number of iterations.
        - The time is used to estimate how many iterations could be done in one minute
      - The second call does this 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)

```
PROGRAM himeno
   INCLUDE "himeno_f77.h"

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

- In the next slides we look at the details of jacobi()

Step 1: Structure of the jacobi routine

- Outer loop is executed fixed number of times
  - loop must be sequential !
- Apply stencil to p to create temporary wrk2
  - residual gosa computed
    - details on the next slide
- Pressure array p updated from wrk2
  - this loopnest can be parallelised
- Outer halo of p is fixed

```
SUBROUTINE jacobi(nn,gosa)

  iteration: 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 iteration

END SUBROUTINE jacobi
```

Step 1: The Jacobi computational kernel

- The stencil is applied to pressure array p
  - 19-point stencil
- Updated pressure values are saved to temporary array wrk2
- Residual value gosa is computed
- This loopnest dominates runtime
  - Can be computed in parallel
  - gosa is reduction variable

```
gosa = 0
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
  - loop variables default private (like OpenMP)
  - scalar variables default private (unlike OpenMP)
  - so clause optional here
    - *advice: use one for clarity*
- copy* data clauses
  - compiler will do automatic analysis
  - explicit clauses will interfere with data directives at next step
    - *advice: only use if compiler over-cautious*

```
gosa1 = 0

!$acc parallel loop reduction(+:gosa1) &
!$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)
   gosa1 = gosa1 + 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 (create clause)

    - The first main code optimisation is 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

g = partitioned loop

G = accelerator kernel

```
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
```

Numbers denote serial loops

source line numbers

```
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_f77_v02.f, 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_f77_v02.f, 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
```

CUDA: k value(s) built from blockIdx.x

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

CUDA: i value(s) built from threadIdx.x

```
ftn-6430 ftn: ACCEL File = himeno_f77_v02.f, Line = 172
  A loop starting at line 172 was partitioned across the thread blocks.

ftn-6509 ftn: ACCEL File = himeno_f77_v02.f, 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_f77_v02.f, Line = 173
  A loop starting at line 173 will be redundantly executed.

ftn-6430 ftn: ACCEL File = himeno_f77_v02.f, Line = 174
  A loop starting at line 174 was partitioned across the 128 threads within a threadblock.
```

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 will also give different results

- ***Advice: make sure the code has checksums, residuals etc. to check for correctness.***
  - *even if code is single precision, try to use double precision for checking.*
    - *globally or at least for global sums and 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 compared to one core.
- **Why?**
  - Only 2% of the GPU time is compute;
    - The rest is data transfer to and from device
- ***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 are copied to 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 it takes negligible compute time, still accelerate for data locality
      - This is a major productivity win for OpenACC compared to low-level languages
        - You can accelerate a loopnest with one directive
        - Don't have to handcode a new CUDA/OpenCL kernel
        - And, remember, the performance of such a kernel is irrelevant

Step 3: Structure of the jacobi routine

- data region spans iteration loop
  - CPU and OpenACC code
  - use explicit data clauses
    - no automatic scoping
    - requires knowledge of app
  - enclosed kernels shouldn't have data clauses for these variables
  - 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)
   iteration: DO loop = 1, nn

! 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 iteration
!$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

- Data region spans both calls to jacobi
  - plus timing calls
- Arrays just need to be copyin now
  - and transfers not timed
- Data region remains in jacobi
  - you can nest data regions
  - arrays now declared present
  - could be copy_or_present
  - advice: present generates runtime error if not present
- Drawback: arrays have to be in scope for this to work
  - may need to unpick clever use of module data

```
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)
   iteration: DO loop = 1, nn

   ENDDO iteration
!$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"
  - need to accelerated loopnests in initmt(), declaring arrays present

- **N.B. Currently no way to ONLY allocated arrays in GPU memory**
  - CPU version is now dead space, but
  - GPU memory is usually the limiting factor, so usually not a problem.

Porting entire application

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

```
PROGRAM himeno

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

   CALL jacobi(nn,gosa)
   cpu1 = gettime()
!$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?

- Look at .lst file
- Should see partitioning between and across threadblocks
  - if not, much of GPU is is being wasted

```
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
```

- Usually want inner loop to be vectorised
  - allows coalesced loading of data from global memory
  - if inner loop is not partitioned over threads in a threadblock...
    - is the loop vectorisable (are there dependencies between loop iterations)?
      - No? You need to rewrite the code (it will probably go faster on the CPU)
        - Can you use a more-explicitly parallel algorithm?
        - Avoid incremented counters (e.g. when packing buffers)
        - Change data layout so inner loop addresses fastest-moving array index
      - Yes? You need to tell the compiler what to do:
        - Put "acc loop vector" directive above the "DO i = ..." statement
- This is the most important optimisation
  - almost guaranteed to give big performance increase
  - other optimisations are trial-and-error and may give no benefits

Advanced performance tuning

- Loop schedule balances lots of parallel threads vs. enough work per thread

```
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
```

- If kmax is small, perhaps need more threads
  - Try collapsing k and j loops to get more loop iterations
    - Put "acc loop collapse(2)" directive above k-loop
  - Collapse can be expensive if compiler has to regenerate k and j
    - integer divides are costly
  - Could instead collapse i and j loops, or all three loops

- Nvidia Fermi and Kepler GPUs have caching
  - Loop blocking can improve cache usage (as for the CPU)
    - Block the loops manually (and use gang, vector clauses to tweak schedule)
    - Can use CCE-specific directives to do this as well

- We'll discuss performance optimisation in more detail in a following lecture

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**

- **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
    - OpenACC is much easier to use, and should get close to CUDA performance
  - Remember Amdahl's Law:
    - speed up the compute of a parallel application, 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 MPI/CAF comms
- **Bottom line:**
  - 5-6x speedup from 7 directive pairs in 200 lines of Fortran
  - compared to the complete CPU

Advanced OpenACC: topics and performance tuning

James Beyer

Timetable

Monday 6th May 2013

8:30	Lecture 1:	Introduction to the Cray XK7	(15)
8:45	Lecture 2:	OpenACC organization (Duncan Poole)	(15)
9:00	Lecture 3:	The OpenACC programming model	(30)
9:30	Lecture 4:	Porting a simple example to OpenACC	(30)
10:00	*break*		*(30)*
10:30	Lecture 5:	Advanced OpenACC	(40)
11:10	Lecture 6:	Using CCE with OpenACC	(25)
11:35	Lecture 7:	OpenACC 2.0 and OpenMP 4.0	(25)
12:00	*close*		

Contents

- **Some more advanced OpenACC topics**
  - the async and cache clauses
- **Then we talk about a few tuning tips for OpenACC**
  - The Golden Rules of Tuning
    - information sources
  - Tuning data locality
  - Tuning kernels
    - correcting obvious scheduling errors
    - advanced schedule tuning (collapse, worker, vector_length clauses)
      - use scalar Himeno code as an example
  - Extreme tuning
    - source code changes
    - reordering data structures
    - using CUDA

OpenACC async clause

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

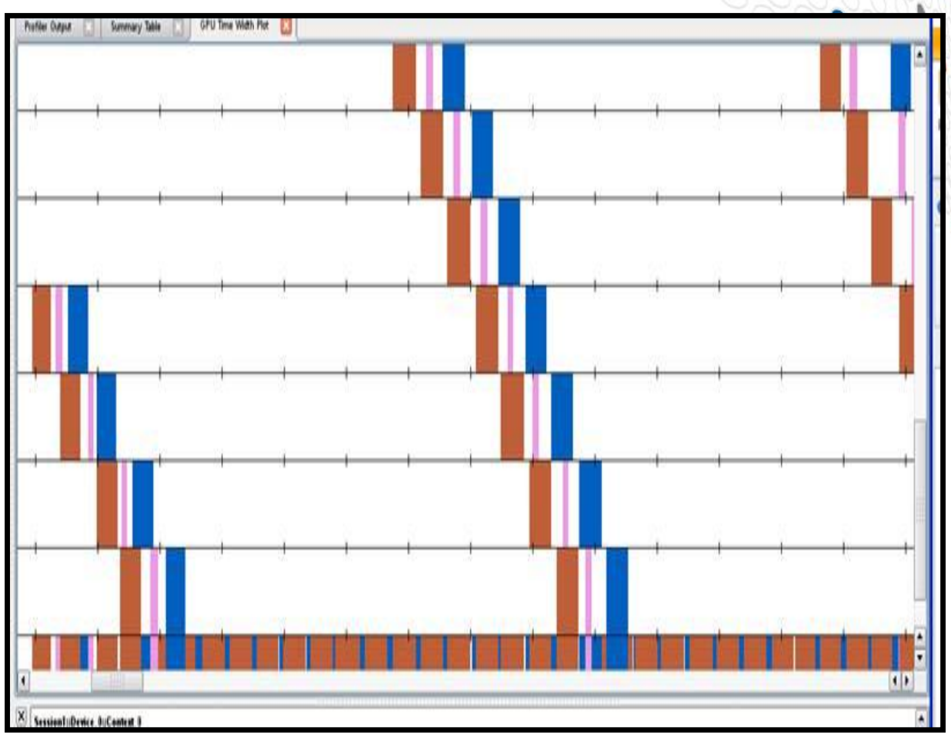
- **!$acc wait: waits for completion of all streams of tasks**
  - !$acc wait(handle) waits for a specified stream to complete
- **Runtime API library functions**
  - can also be used to wait or test for completion

OpenACC async clause

```
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
```

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

OpenACC async results



- **Execution times (on Cray XK6):**
  - 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

Using the cache clause

- **Performance-tuning clause**
  - Don't worry about this when first accelerating a code
  - Apply it later to the slowest kernels of working OpenACC port

- **Suggests that compiler could place data into software-managed cache**
  - e.g. threadblock-specific "shared" memory on Nvidia GPU
  - No guarantee it makes the code faster
    - could conflict with automatic caching done by hardware and/or runtime
- **Clause inserted inside kernel**
  - i.e. inside all the accelerated loops
- **Written from perspective of a single thread**
  - Compiler pools statements together for threadblock
  - Limited resource: use sparingly and only specify what's needed
  - Any non-loop variables should be compile-time parameters (CCE)

cache clause examples

- **Example 1:**
  - loop-based stencil
  - inner loop 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
```

cache clause examples

- **Example 2**
  - from "man openacc.examples"
  - multidimensional loopnest
    - stencil only in i,j directions
  - same principle, but...
    - you need to tile the loopnest
    - two options currently:
      - do it explicitly
        - DO jb = 1,N,JBS
        - DO j = jb,MIN(jb+JBS-1,N)
        - and similarly for i
      - use CCE directives, as right
    - OpenACC v2.0 will ease this:
      - tile clause for loop directive
      - more on this later in course

```
!$acc loop gang
DO k = 1,N
!dir$ blockable( i, j )
!$acc loop worker
!dir$ blockingsize ( 16 )
  DO j = 1,N
!$acc loop vector
!dir$ blockingsize ( 64 )
    DO i = 1,N
!$acc cache( A(i,j,k), &
!$acc        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 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)
      - loopmark files (-hlist=a) for CCE; -Minfo=accel for PGI
    - Runtime commentary (CCE only: CRAY_ACC_DEBUG=1 or 2 or 3)
    - Code profiling
      - CrayPAT
      - 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
  - 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
  - Beware of GPU memory allocation overheads
    - if a routine using big temporary arrays is called many times, even create clause can have a big overhead
    - 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)

Kernel optimisation

- **Next optimisation: make sure all the kernels vectorise**
  - How can I tell if this is a problem?
    - if a kernel is surprisingly slow on accelerator
      - in a wildly different place in the the profile compared to running on CPU
    - examine the loopmark compiler commentary files
  - loop iterations should be divided over both the threads in a threadblock (vector) and over the threadblocks (gang)
    - CCE: you should see either:
      - If a single loop is divided over both levels of parallelism, look for: Gg
      - If two different loops divided, look for G and 2 g-s (maybe with numbers between)
  - generally want to vectorise the innermost loop
    - usually fastest-moving array index, for coalescing
  - if not, can the inner loop be vectorised?
    - i.e. 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 also

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 vectorisable, guide the compiler**
  - a gentle hint:
    - put "acc loop independent" directive above this loop
    - could also use CCE directive "!dir$ concurrent"
      - see "man intro_directives" for details
  - a direct order:
    - put "acc loop vector" directive above this loop
  - check the code is still correct and running faster, though:
    - the compiler might not be vectorising for a good reason

- **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
  - 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 (which is slower)
  - probably also want to reorder array c for speed
    - c(i,m) gives much coalesced memory accesses
    - want vector index to be fastest-moving index

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

It's all vectorizing, 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

Advanced loop scheduling

- **OpenACC loop schedules are limited by the loop bounds**
  - at least with the current implementation in CCE
  - one loop's iterations are divided over gangs
  - another loop's iterations are divided over threads in a threadblock
- **So...**
  - "tall, skinny" loopnests (j=1:big; i=1:small) won't schedule well
    - 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
    - want lots of threadblocks to swap amongst SMs
- **What can we do?**
  - collapse clause is way of increasing flexibility
    - the compiler may use this automatically (look for C in loopmark)
    - no guarantee that it is faster
      - e.g. index rediscovery requires expensive integer divisions
    - need perfectly nested loops for this to work
  - worker clause can also do this

Using the collapse clause

- **Consider a three-level loopnest (i inside j inside k)**
  - needs to be perfectly nested to use collapse
  - Collapse all three loops and schedule across GPU
    - "acc parallel loop collapse(3) gang worker vector" above k-loop
      - probably don't need "gang worker vector" here
  - 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
  - Schedule outer two loops over the threadblocks
    - "acc parallel loop collapse(2) gang" above k-loop
    - "acc loop vector" above i-loop
  - Schedule outer two loops together over entire GPU
    - "acc loop collapse(2) gang worker vector" above k-loop
    - "acc loop seq" above i-loop
  - Schedule k-loop and i-loop together over entire GPU
    - collapsed loops must be perfectly nested; you'll need to reorder the code

workers or vectors?

- **kernel threadblocks are scheduled on SMs**
  - executed as "warps" i.e. vector instructions of length 32
  - threads-per-threadblock>32 automatically decomposed into warps

- **OpenACC makes distinction explicit**
  - worker refers to whole warps (i.e. sets of vector instructions)
    - can be generated explicitly by the user using "!$acc loop worker"
  - vector refers to threads within a warp
    - can be generated automatically by the compiler/runtime
      - vector_length > 32 automatically decomposes into (vector_length/32) workers

- **CCE: only allows one of the above**
  - If you don't specify "!$acc loop worker"
    - vector_length (default 128) automatically partitioned into workers
    - num_workers works the same
  - If you specify "!$acc loop worker"
    - default, or vector_length explicitly set
      - num_workers implicitly set to (vector_length/32)
      - vector_length implicitly set to 32 (see loopmark for information)
    - num_workers explicitly set
      - vector_length set to 32
    - num_workers and vector_length>32 explicitly set
      - Compiler warning that vector_length value is being overridden and set to 32

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
!$acc loop seq
   DO j = 1,N
!$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 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
```

workers or vectors (contd)?

- **So when might we use "!$acc loop worker"?**

- **Perfectly nested loops with one or more low tripcounts**
  - probably better to use the collapse clause
    - e.g. "!$acc loop collapse(2) vector"
    - we'll see this for scalar Himeno shortly

- **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

Extreme tuning

- **You've tried tuning with OpenACC clauses**
  - but you think kernel performance can still be improved
  - (and this kernel is the 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
  - mixed languages
    - You could handtune the slow kernel in CUDA
    - OpenACC allows interoperability with CUDA (i.e. sharing data)
    - Following slides give a very simple example

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 fission 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:**
  - Nmax particles each have Smax internal properties
    - code separately combines the internal properties together 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)

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

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

Using CCE with OpenACC

Timetable

Monday 6th May 2013

8:30	Lecture 1:	Introduction to the Cray XK7	(15)
8:45	Lecture 2:	OpenACC organization (Duncan Poole)	(15)
9:00	Lecture 3:	The OpenACC programming model	(30)
9:30	Lecture 4:	Porting a simple example to OpenACC	(30)
10:00	*break*		*(30)*
10:30	Lecture 5:	Advanced OpenACC	(40)
11:10	Lecture 6:	Using CCE with OpenACC	(25)
11:35	Lecture 7:	OpenACC 2.0 and OpenMP 4.0	(25)
12:00	*close*		

Contents

- **Cray Compilation Environment (CCE)**
  - What does CCE do with X?
  - -hacc_model=
  - Extensions
    - Structure shaping
    - Deep copy
    - Selective deep copy

OpenACC in CCE

- **man intro_openacc**
- **Which module to use**
  - craype-accel-nvidia20
  - craype-accel-nvidia35
- **Forces dynamic linking**
- **Single object file**
- **Whole program**
- **Messages/list file**
- **Compiles to PTX not cuda**
- **Debugger sees original program not cuda intermediate**

What does CCE do with OpenACC constructs (1)

- **Parallel/kernels**
  - Flatten all calls
  - Package code for kernel
  - Insert data motion to and from device
    - Clauses
    - Autodetect
  - Insert kernel launch code
  - Automatic vectorization is enabled
- **Kernels**
  - Identify kernels
- **Loop**
  - Gang
    - Thread Block (TB)
  - Worker
    - warp
  - Vector
    - Threads within a warp or TB
  - Automatic vectorization is enabled
  - Collapse
    - Will only rediscover indices when required
  - Independent
    - Turns off safety/correctness checking for work-sharing of loop
  - Reduction
    - Nontrivial to implement
    - Does not use multiple kernels
    - All loop directives within a loop nest must list to reduction if applicable

What does CCE do with OpenACC constructs (2)

- **Data**
  - *clause(object list)*
  - create
    - allocate at start
    - register in “present-table”
    - de-allocate at exit
  - copy, copyin, copyout
    - “create” plus data copy
  - present
    - Abort at runtime if object is not in “present table”.
  - present_or_copy, present_or_copyin, present_or_copyout, present_or_create
  - deviceptr
    - Send address directly to kernel without translation.

- **Update**
  - Implicit !$acc data present(obj)
  - For known contiguous memory
    - Transfer (Essentially a CUDA memcpy)
  - Not contiguous memory
    - Pack into contiguous buffer
    - Transfer contiguous
    - Unpack from contiguous buffer

What does CCE do with OpenACC constructs (3)

- **Cache**
  - Create shared memory “copies” of objects
  - Generate copy into shared memory objects
  - Generate copy out of shared memory objects
  - Release the shared memory

Extended OpenACC 1.0 runtime routines

```
/* takes a host pointer */
void* cray_acc_create( void* , size_t );
void  cray_acc_delete( void* );
void* cray_acc_copyin( void*, size_t );
void  cray_acc_copyout( void*, size_t );
void  cray_acc_updatein( void*, size_t );
void  cray_acc_updateout( void*, size_t );
int   cray_acc_is_present( void* );
int   cray_acc_is_present_2( void*, size_t);
void *cray_acc_deviceptr( void* );

/* takes a device and host pointer */
void  cray_acc_memcpy_device_host( void*, void*, size_t );
/* takes a host and device pointer */
void  cray_acc_memcpy_host_device( void*, void*, size_t );
```

```
/* Takes a pointer to an implementation defined type */
bool cray_acc_get_async_info( void *, int )
```

```
/* takes a device and host pointer */
void  cray_acc_memcpy_device_host( void*, void*, size_t );
/* takes a host and device pointer */
void  cray_acc_memcpy_host_device( void*, void*, size_t );
```

Partitioning clause mappings

1. !$acc loop gang : across thread blocks
2. !$acc loop worker : across warps within a thread block
3. !$acc loop vector : across threads within a warp

1. !$acc loop gang : across thread blocks
2. !$acc loop worker vector : across threads within a thread block

1. !$acc loop gang : across thread blocks
2. !$acc loop vector : across threads within a thread block

1. !$acc loop gang worker: across thread blocks and the warps within a thread block
2. !$acc loop vector : across threads within a warp

1. !$acc loop gang vector : across thread blocks and threads within a thread block

1. !$acc loop gang worker vector : across thread blocks and threads within a thread block

Partitioning clause mappings (cont)

You can also force things to be within a single thread block:

1. !$acc loop worker : across warps within a single thread block
2. !$acc loop vector : across threads within a warp

1. !$acc worker vector : across threads within a single thread block

1. !$acc vector : across threads within a single thread block

-hacc_model options

- **auto_async_(none | kernel | all)**
  - Compiler automatically adds some asynchronous behavior
  - Only overlaps host and accelerator
  - No automatic overlap of different accelerator constructs (single stream)
  - May require some explicit user waits
    - Host_data
- **[no_]fast_addr**
  - Uses 32 bit variables/calculations for index expressions
  - Faster address computation
  - Fewer registers
- **[no_]deep_copy**
  - Enable automatic deep copy support

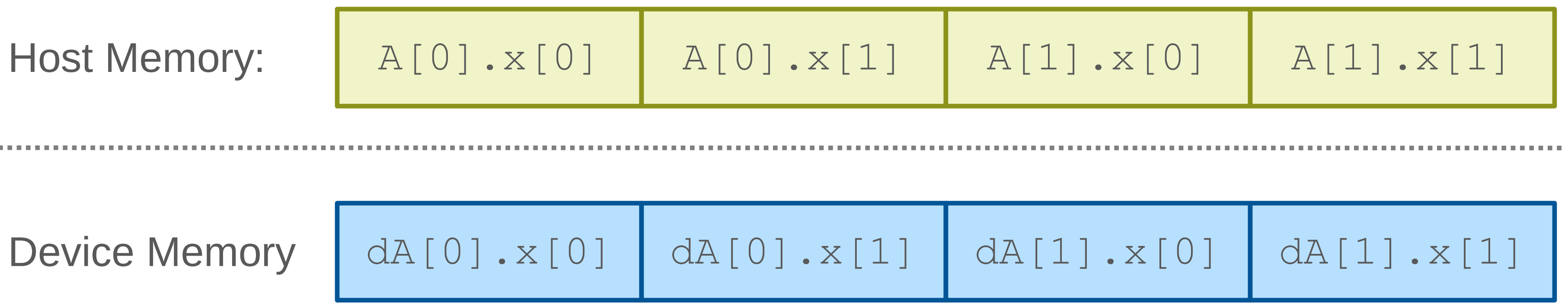
Extensions

- Deep copy
- Structure shaping
- Selective deep copy

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])
```

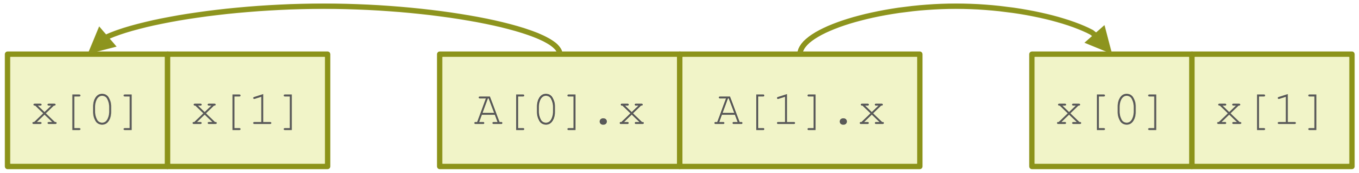


Challenges with pointer indirection

- Non-contiguous transfers
- Pointer translation

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

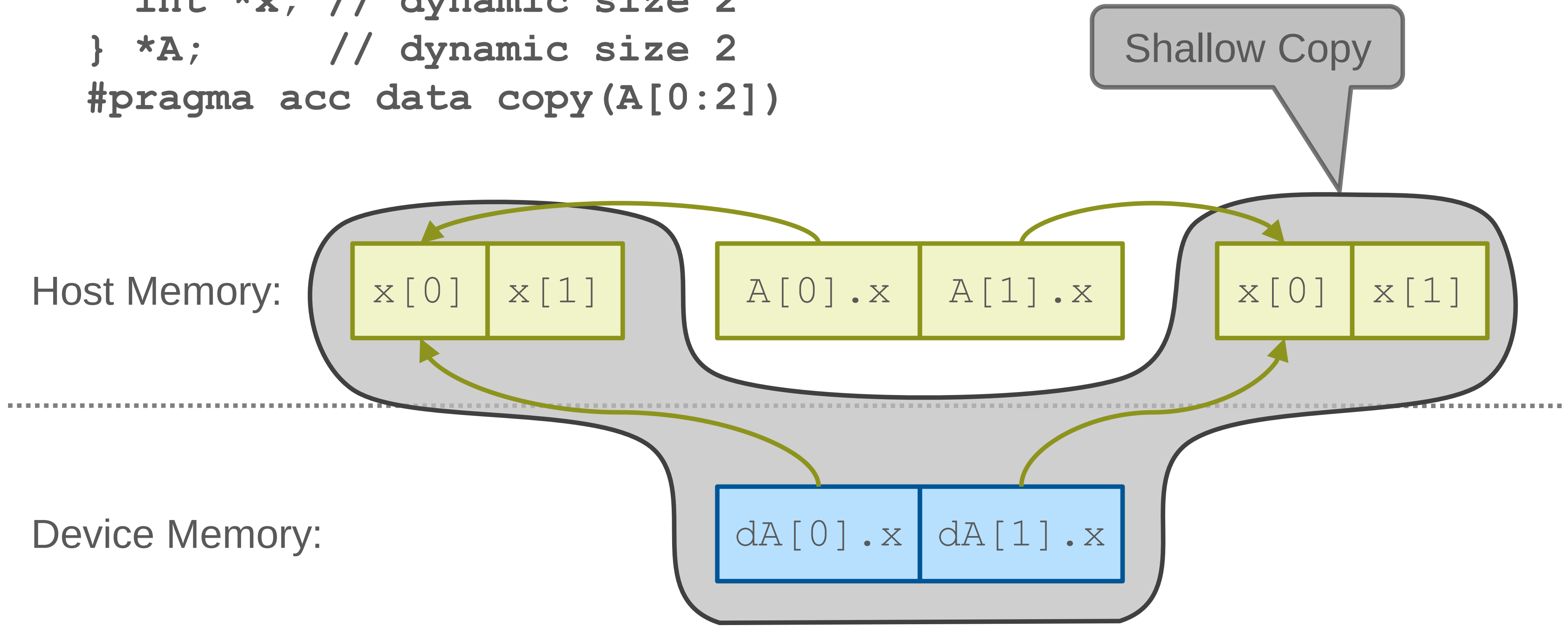
Host Memory:



Challenges with pointer indirection

- Non-contiguous transfers
- Pointer translation

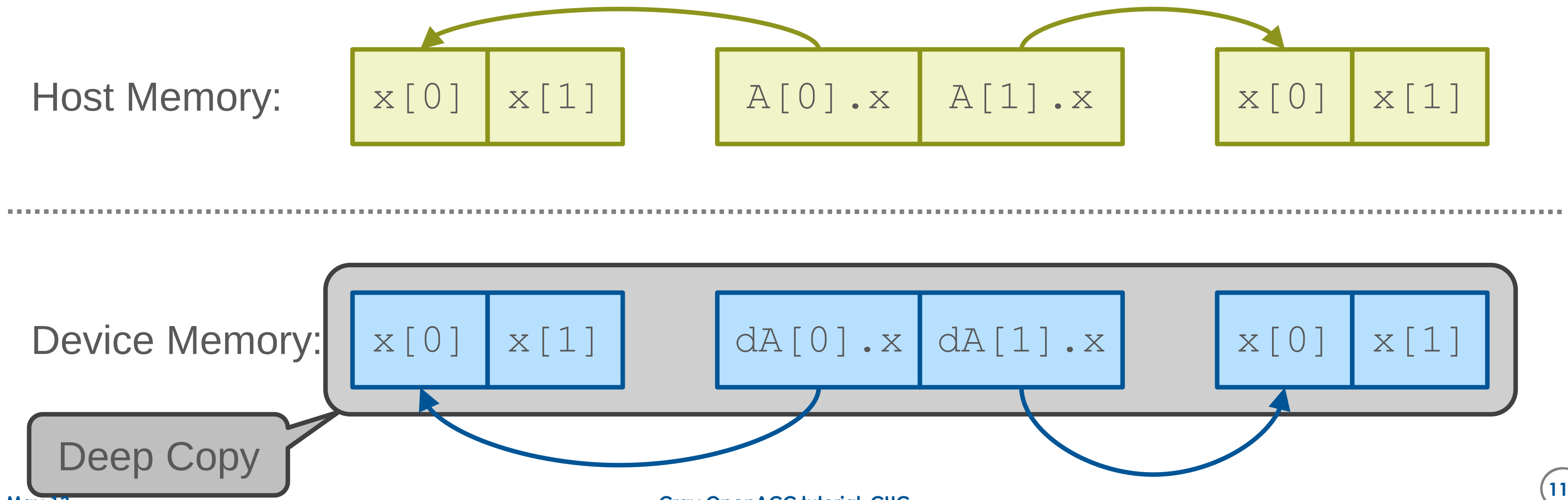
```
struct {
  int *x; // dynamic size 2
} *A;     // dynamic size 2
#pragma acc data copy(A[0:2])
```



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++
- Portable in OpenACC 2.0, but not usually practical

Automatic Fortran deep-copy

```
type A_t
   integer,allocatable :: x(:)
end type A_t
...
type(A_t),allocatable :: A(:)
...
! shallow copy with -hacc_model=no_deep_copy (default)
!    deep copy with -hacc_model=deep_copy
!$acc data copy(A(:))
```

- No aliases on the accelerator
- Must be contiguous
- On or off – no “selective” deep copy
- Only works for Fortran

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
```

Member Shape Status

- **Library**
  - Support for type descriptors
- **Compiler**
  - Automatic generation of type descriptors for Fortran
    - Compiler flag to enable/disable deep copy
    - Released in CCE 8.1
    - Significant internal testing, moderate customer testing
  - Directive-based generation of type descriptors for C/C++
    - Planned for release in CCE 8.2
    - Limited preliminary internal testing
- **Language**
  - Committee recognizes the utility and need
  - Will revisit after OpenACC 2.0

OpenACC 2.0 & OpenMP 4.0

James C. Beyer

Timetable

Monday 6th May 2013

8:30	Lecture 1:	Introduction to the Cray XK7	(15)
8:45	Lecture 2:	OpenACC organization (Duncan Poole)	(15)
9:00	Lecture 3:	The OpenACC programming model	(30)
9:30	Lecture 4:	Porting a simple example to OpenACC	(30)
10:00	*break*		*(30)*
10:30	Lecture 5:	Advanced OpenACC	(40)
11:10	Lecture 6:	Using CCE with OpenACC	(25)
11:35	Lecture 7:	OpenACC 2.0 and OpenMP 4.0	(25)
12:00	*close*		

Contents

- OpenACC 2.0
  - New directives
  - Status
- OpenMP 4.0 accelerator support
  - New directives
  - Status
- Differences between OpenACC and OpenMP
- Usage/Porting tips

OpenACC 2.0 key features

- Procedure calls, separate compilation
- Nested parallelism
- Device-specific tuning, multiple devices
- Data management features and global data
- Multiple host thread support
- Loop directive additions
- Asynchronous behavior additions
- New API routines
- Default(none)

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

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.

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 );
 }
```

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])
}
```

Multiple host thread support

- Share the device context
- Share the device data
- Can create race conditions
- present_or_copy is your friend

- This is what Cray has always done, now it is well defined.

Loop directive additions

- loop gang may not contain loop gang
- loop worker may not contain loop gang, worker
- loop vector may not contain gang, worker, vector
- added loop auto (compiler selects)

- Tile clause
  - tile(16,16) gang vector
  - !$acc loop tile(64,4) gang vector

```
do i = 1, n
   do j = 1, m
      a(j,i) = (b(j-1,i)+b(j+1,i)+ &
               b(j,i-1)+b(j,i+1))*0.25
   enddo
enddo
```

Asynchronous behavior additions

- Allow async clause on wait directive
  - Join two async streams without waiting on host
  - !$acc wait(1) async(2)
    - All previous work on async(1) must complete before any new work added to async(2) can execute
    - Adds a join with async(1) in the async(2) queue
- Allow wait clause on any directive that supports async
  - Parallel, kernels, update, …
- Allow multiple async identifiers in a wait directive/clause

New API routines

acc_copyin(ptr, bytes)
acc_create(ptr, bytes)
acc_copyout(ptr, bytes)
acc_delete(ptr, bytes)
acc_is_present(ptr, bytes)
acc_update_device(ptr, bytes)
acc_update_local(ptr, bytes)
acc_deviceptr(ptr)
acc_hostptr(devptr)
acc_map_data(devptr, hostptr, bytes)
acc_unmap_data(hostptr)

Default(none)

- No implicit data scoping/mapping will be performed
- It is an error if a non-predetermined variable is not in a data clause

OpenACC 2.0 status

- All major features accepted
- Closing in on the final feature set
- Plan release for ISC'13
  - Biggest risk is the editor's time

- A common directive programming model for shared memory systems
- Announced 15yrs ago
- Works with Fortran, C, C++
- Current version 3.1 (July 2011)
- Accelerator version 4.0 (?? 2013)
- Compiler support
  - http://openmp.org/wp/openmp-compilers/

OpenMP 4.0 accelerator 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
- **Array sections**

OpenMP 4.0 status

- **Accelerator support version 1 accepted**
- **Currently in comment period**
- **Language committee members doing section by section review**
- **Hoping for a May release, not very likely**
- **There were several compromises in this version**
  - Bitwise copies for both language classes
    - No auto-deep copy in fortran
    - No constructors in C++ for data motion
  - Single type of accelerator per compile
  - …

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	**Update**
Wait	
Declare	**Declare**

OpenACC compared to OpenMP continued

OpenACC

- enter data
- exit data
- data api
- routine
- async wait
- parallel in parallel
- tile

OpenMP

-
-
-
- declare target
-
- Parallel in parallel or team
-

OpenACC compared to OpenMP continued

OpenACC

-
-
-
-
-
-
-
-
-

OpenMP

- **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
- **Is this the best solution?**

Porting code to OpenACC (kernel level)

- **Identify parallel opportunities**
- **For each parallel opportunity**
  - Add OpenACC Parallel Loop(s)
  - Verify correctness
  - Avoid data clause when possible, use present_or_* when required
- **Optimize "kernel" performance**
  - Add additional acc loop directives
  - Add tuning clause/directives (collapse, cache, num_gangs, num_workers, vector_length, …)
  - Algorithmic enhancements/code rewrites
- **Try fast address option**

Porting code to OpenACC (application level)

- **Add data regions/updates**
  - Try to put data regions as high in the call chain as profitable
  - Working with one variable at a time can make things more manageable
  - To identify data correctness issues can add excessive updates and remove them verifying correctness.
- **Try auto async all**
  - Auto async kernel is default
- **Add async clauses and waits**
  - If synchronization issues are suspected, try adding extra waits and slowly remove them.

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 gaps**
- **Differences are significant enough that OpenACC may never fold back into OpenMP**
  - OpenACC aims for portable performance
  - OpenMP aims for programmability

CRAY
CRAY XC30

ECOphlex
CRAY XE6

ECOphlex
CRAY XK7