

An Introduction to OpenACC

Alistair Hart
Cray Exascale Research Initiative Europe



Piz Daint



Top500 List - November 2013

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
6	Swiss National Supercomputing Centre (CSCS) Switzerland	Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115984	6271.0	7788.9	2325

The Green500 List

Listed below are the November 2013 The Green500's energy-efficient supercomputers ranked from 1 to 10.

Green500 Rank	MFLOPS/W	Site*	Computer*	Total Power (kW)
1	4,503.17	GSIC Center, Tokyo Institute of Technology	TSUBAME-KFC - LX 1U-4GPU/104Re-1G Cluster, Intel Xeon E5-2620v2 6C 2.100GHz, Infiniband FDR, NVIDIA K20x	27.78
2	3,631.86	Cambridge University	Wilkes - Dell T620 Cluster, Intel Xeon E5-2630v2 6C 2.600GHz, Infiniband FDR, NVIDIA K20	52.62
3	3,517.84	Center for Computational Sciences, University of Tsukuba	HA-PACS TCA - Cray 3623G4-SM Cluster, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband QDR, NVIDIA K20x	78.77
4	3,185.91	Swiss National Supercomputing Centre (CSCS)	Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Level 3 measurement data available	1,753.66

COMPUTE

STORE

ANALYZE

OpenACC at SC13



COMPUTE

STORE

ANALYZE

Timetable

- 09:00 Lecture: Introduction to the Cray XC30 L1
- 09:30 Lecture: OpenACC 1: a first look L2
- 10:00 Lecture: Using the Cray XC30 L3
- 10:10 Practical: Running a very simple job same
 - 10:15 ----- Break -----

- 10:45 Live Demo: Porting a simple code L4
- 11:15 Lecture: OpenACC 2: more features L5
- 11:45 Practical: The simple code same
 - 12:15 ----- Lunch -----

- 13:15 Lecture: Debugging (by CSCS) L6
- 13:45 Lecture: OpenACC 3: scheduling and tuning L7
- 14:15 Example: Tuning a simple code L8
- 14:30 Lecture: OpenACC 4: handling asynchronicity L9
- 14:50 Example: Porting a parallel code L10
 - 15:15 ----- Break -----

- 15:45 Lecture: OpenACC 5: advanced topics L11
- 16:05 Example: Porting a larger code L12
- 16:30 Practical: The larger code script
 - 17:15 Lecture: OpenACC 6: the future L13
 - 17:30 ----- 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 in using OpenACC directives
 - with some hints and tips along the way
 - To introduce you to profiling tools
 - to understand and tune OpenACC performance
 - to help you understand a real application to start OpenACC-ing...
- **The idea is to equip you with the knowledge to develop applications that run efficiently on parallel hybrid supercomputers**
 - not just on single GPUs

Inside the Cray XC30 and the Nvidia Kepler K20X GPU

Alistair Hart
Cray Exascale Research Initiative Europe



COMPUTE | STORE | ANALYZE

The XC30 architecture

- **Node architecture:**

- One Intel Sandybridge CPU (8 cores)
- One Nvidia GPU
 - Kepler K20x
 - 2688 + 896 cores, 1.3 TFlop/s DP, 6GB memory

- **Cray Aries interconnect**

- one per blade, shared between four nodes
- Dragonfly network topology
 - 3 levels of all-to-all networks (two copper, one optical)
- high bandwidth/low latency scalability

- **Fully integrated/optimised/supported**

- Tight integration of GPU and NIC drivers

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)

A quick GPU refresher

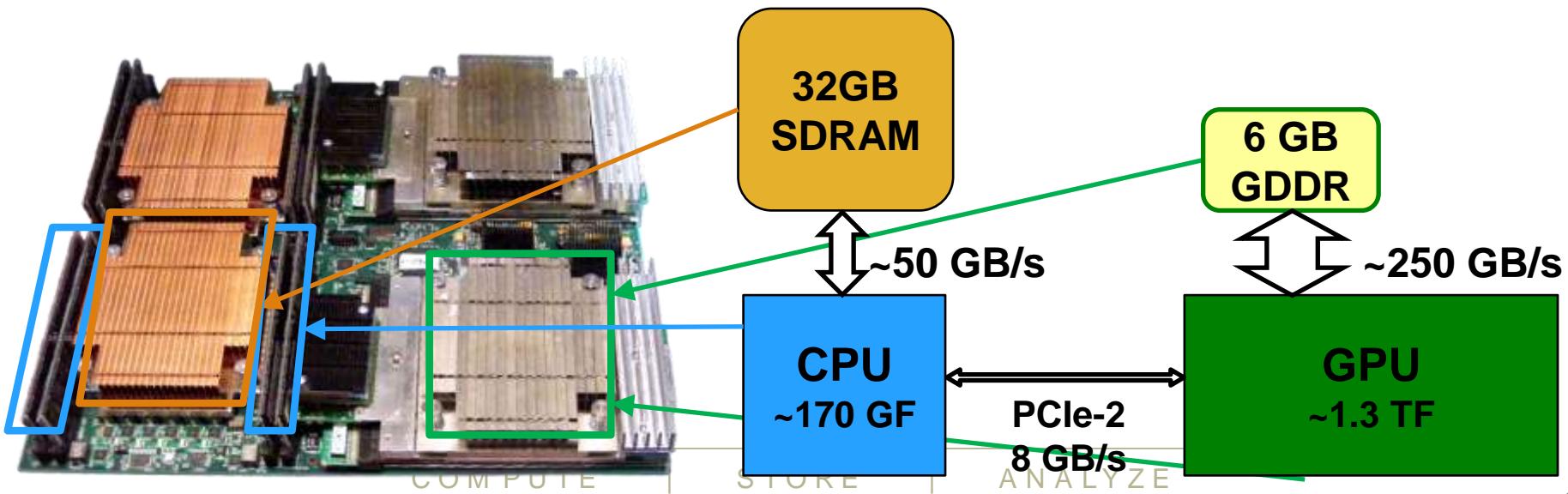
How fast are current GPUs?

- What should you expect?

- Cray XC30:
 - Flop/s: GPU ~8x faster than single, whole CPU (8 cores)
 - Memory bandwidth: GPU ~5x faster than CPU
- These ratios are going to be similar in other systems

- Plus, it is harder to reach peak performance on a GPU

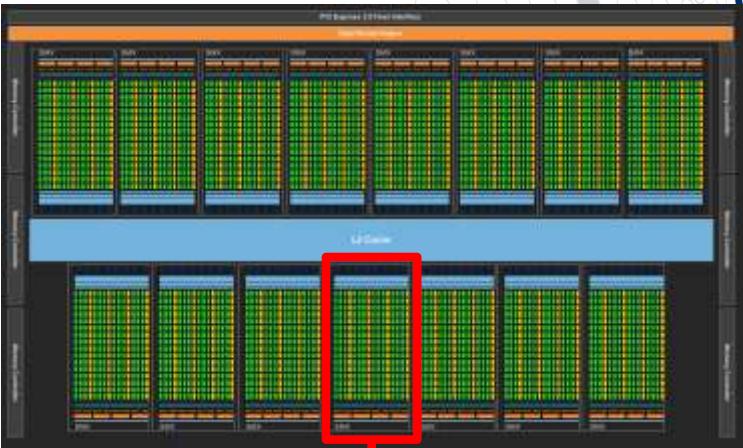
- Your code needs to fit the architecture
- You also need to factor in data transfers between CPU and GPU



Nvidia K20X Kepler architecture

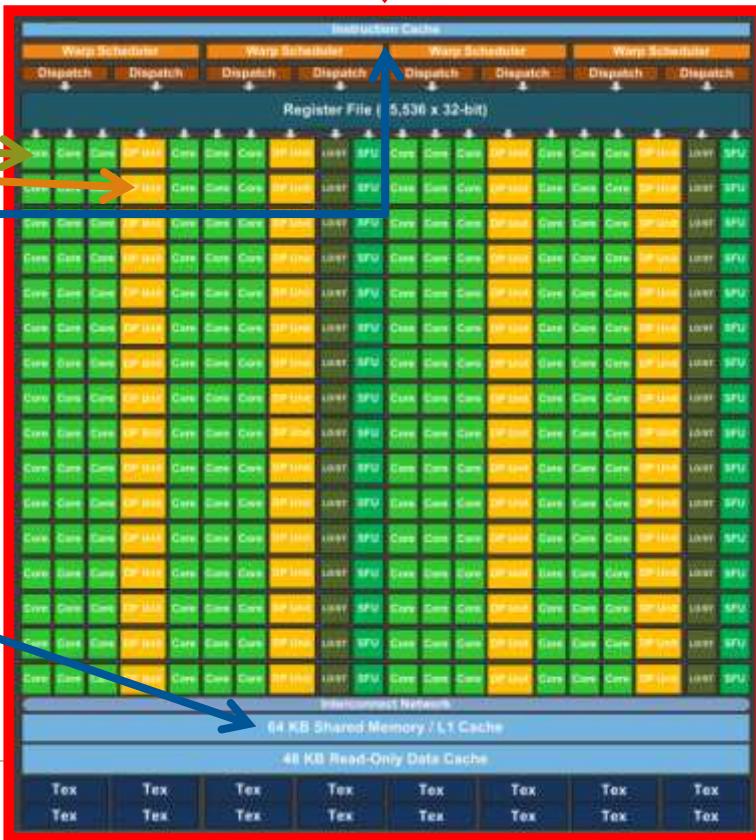
• Global architecture

- 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

COMPUTE STORE

Execution and Memory models

- **GPUs have a host-directed execution model**
 - GPUs "devices" require a CPU "host", which directs execution:
 - Manages host and device memory
 - Launches computational "kernels" to execute on the GPU
 - Initiates data transfers between memory spaces
 - Handles synchronisation of these events
- **GPUs have a weak memory model**
 - host and device have separate memories
 - different memory addresses, different data
 - there is no automatic synchronisation of the memory spaces
 - all synchronisation must be done explicitly by the host
 - directed either by the user or by the runtime (the compiler may help)

Program execution on a GPU

- **Kernels are launched by CPU to execute on GPU**
 - CUDA programming model:
 - parallel work divided between large number of lightweight **threads**
 - threads are divided into logical sets (known as **threadblocks**)
- **The GPU runtime schedules these on the GPU hardware**
 - each **threadblock** executes on a single SM
 - threads in threadblock have some shared memory
 - you can synchronise the threads in a threadblock
 - different threadblocks may execute on different SMs
 - GPU runtime dynamically swaps threadblocks in and out of execution
 - to hide memory latency
 - you cannot synchronise between threadblocks within kernel
- **The kernels launch asynchronously**
 - global synchronisation is handled by the host at the end of the kernel

What CUDA doesn't tell you (upfront)

- **Threads are not created equal**
 - The SM is really a vector processor of width 32
 - groups of 32 cores act in lockstep, rather than independently
 - shares a single instruction stream with single program counter
 - Threads within a threadblock are divided into sets of 32 (**warps**)
 - each warp executes in SIMD fashion using 32 cores of SM
 - if a threadblock contains multiple warps, these are executed in turn
 - i.e. if there are more than 32 threads in the threadblock
 - Memory loads/stores are also done on a per-warp basis
 - Loading/storing 32 consecutive memory addresses at once
- **So, really, the compiler is implementing your code using vector instructions**
 - This is not explicit in the CUDA programming model, but is crucial to gaining good performance from a GPU
 - whichever programming model you are using (it's a hardware thing)

What does this mean for the programmer?

- You need **a lot of parallel tasks** (i.e. loop iterations) to keep GPU busy
 - Each parallel task maps to a thread in a threadblock
 - You need a lot of threadblocks per SM to hide memory latency
 - Not just 2688 parallel tasks, but 10^4 to 10^6 or more
 - This is most-likely in a loop-based code, treating iterations as tasks
 - OpenACC is particularly targeted at loop-based codes
- Your inner loop must **vectorise** (at least with vector length of 32)
 - So we can use all 32 threads in a warp with shared instruction stream
 - Branches in inner loop are allowed, but not too many
- Memory should be accessed in the correct order
 - Global memory access is done with (sequential) vector loads
 - For good performance, want as few of these as possible
 - so all the threads in warp should collectively load a contiguous block of memory at the same point in the instruction stream
 - This is known as "**coalesced memory access**"
 - So vectorised loop index should be fastest-moving index of each array

COMPUTE | STORE | ANALYZE

What does this mean for the programmer?

- No internal mechanism for synchronising between threadblocks
 - Synchronisation must be handled by host
 - So reduction operations are more complicated
 - even though all threadblocks share same global memory
 - Fortunately launching kernels is cheap
 - GPU threadteams are "lightweight"
- Data transfers between CPU and GPU are very expensive
 - You need to concentrate on "**data locality**" and avoid "**data sloshing**"
 - Keeping data in the right place for as long as it is needed is crucial
 - You should port as much of the application as possible
 - This probably means porting more than you expected

OpenACC suitability

- **Will my code accelerate well with OpenACC?**

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

COMPUTE | STORE | ANALYZE

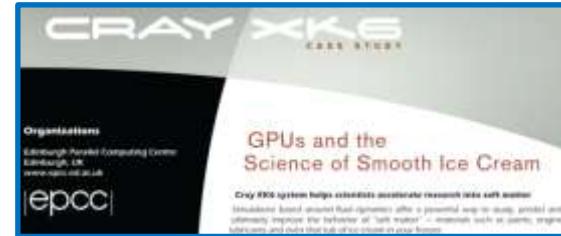
So...

- **GPUs can give very good performance**
 - but you need to be aware of the underlying architecture
 - porting a real application to GPU(s) requires some hard work
 - Amdahl says you need to port a lot of the profile to see a speed-up
 - bad news: to see 10x speedup, need to port at least 90% of the application profile
 - good news: if profile very peaked, 90% of time may be spent in, say, 40% of code
 - even before you worry about the costs of data transfers
- **A good programming model and environment**
 - helps bridges the gap between **peak** and **achievable** performance

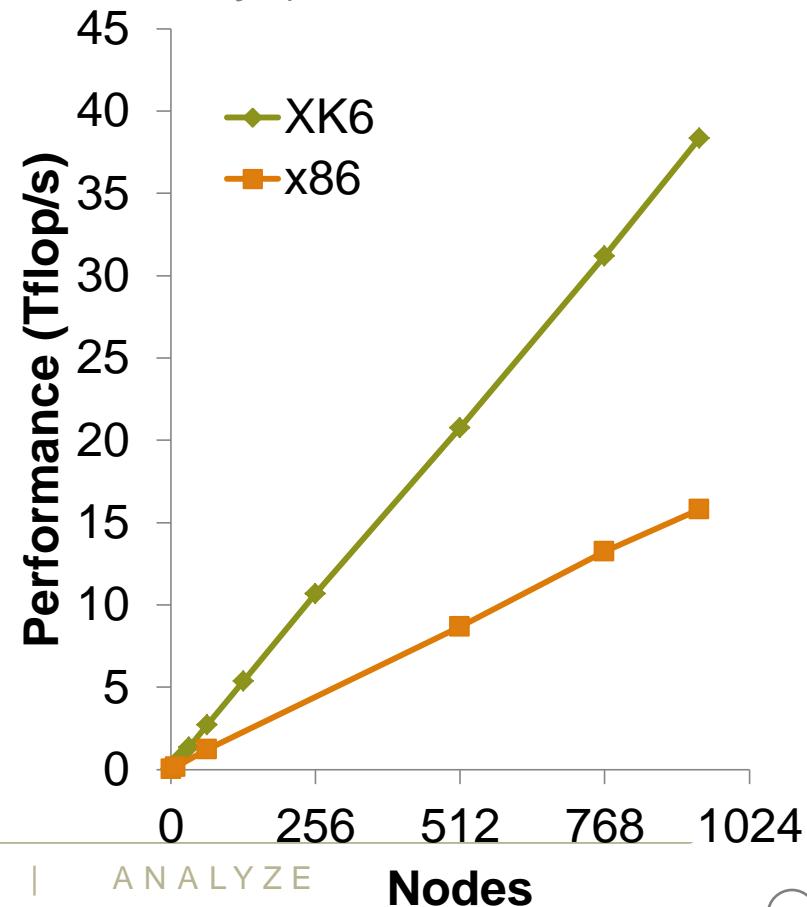
Accelerator programming

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

CUDA for experts



- If you work hard, you can get good parallel performance
- Ludwig Lattice Boltzmann code rewritten in CUDA
 - Reordered all the data structures (structs of arrays)
 - Pack halos on the GPU
 - Streams to overlap compute, PCIe comms, MPI halo swaps
- 10 cabinets of Cray XK6
 - 936 Nvidia Fermi GPUs (nodes)
- Only 4% deviation from perfect weak scaling between 8 and 936 GPUs.
- But...
 - Most applications don't have this level of expert developer support



COMPUTE

STORE

ANALYZE

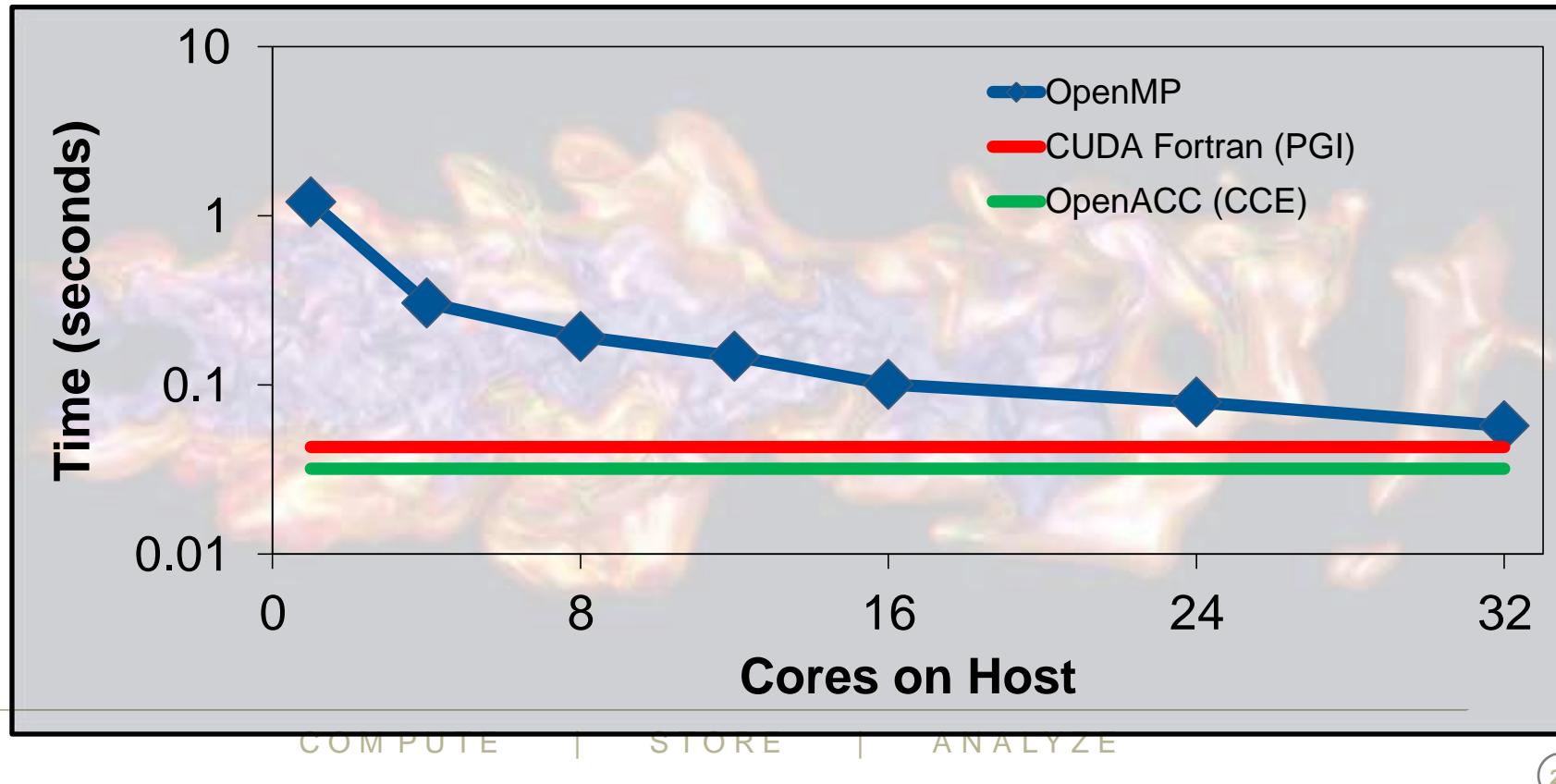
Directive-based programming

Directives provide a high-level alternative

- + **Based on original source code (Fortran, C, C++)**
 - + Easier to maintain/port/extend code
 - + Users with OpenMP experience find it a familiar programming model
 - + Compiler handles repetitive coding (cudaMalloc, cudaMemcpy...)
 - + Compiler handles default scheduling; user tunes only where needed
- **Possible performance sacrifice**
 - Important to quantify this
 - Can then tune the compiler
 - Small performance sacrifice is an acceptable
 - trading-off portability and productivity against this
 - after all, who handcodes in assembler 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):



Strategic risk factors of OpenACC

- Will there be machines to run my OpenACC code on?
 - Now? Lots of Nvidia GPU accelerated systems
 - Cray XC30s and XK7s, plus other vendors (OpenACC is multi-vendor)
 - Future? OpenACC can be targeted at other accelerators
 - PGI and CAPS already target Intel Xeon Phi, AMD GPUs
 - Plus you can always run on CPUs using same codebase
- Will OpenACC continue?
 - Support? Cray, PGI, CAPS committed to support. Now gcc as well.
 - Lots of big customer pressure to continue to run OpenACC
 - Develop? OpenACC committee now 18 partners
 - v2.0 finalised in 2013, now working on next version (2.1 or 3.0)
- Will OpenACC be superseded by something else?
 - Auto-accelerating compilers? Yes, please! But never managed before
 - Data locality adds to the challenge
 - OpenMP accelerator directives? Immature at the moment
 - OpenACC work not wasted: thinking takes more time than coding
 - Very similar programming model; can transition when these release if wish
 - Cray (co-chair), PGI very active in OpenMP accelerator subcommittee

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?
 - How do I port a real-sized application?
 - Performance tuning and advanced topics
- **It will assume you know**
 - A little bit about GPU architecture and programming
 - SMs, threadblocks, warps, coalescing
- **It will help if you know**
 - The basic idea behind OpenMP programming
 - but this is not essential

The OpenACC programming model:

1. a first look

Alistair Hart

Cray Exascale Research Initiative Europe

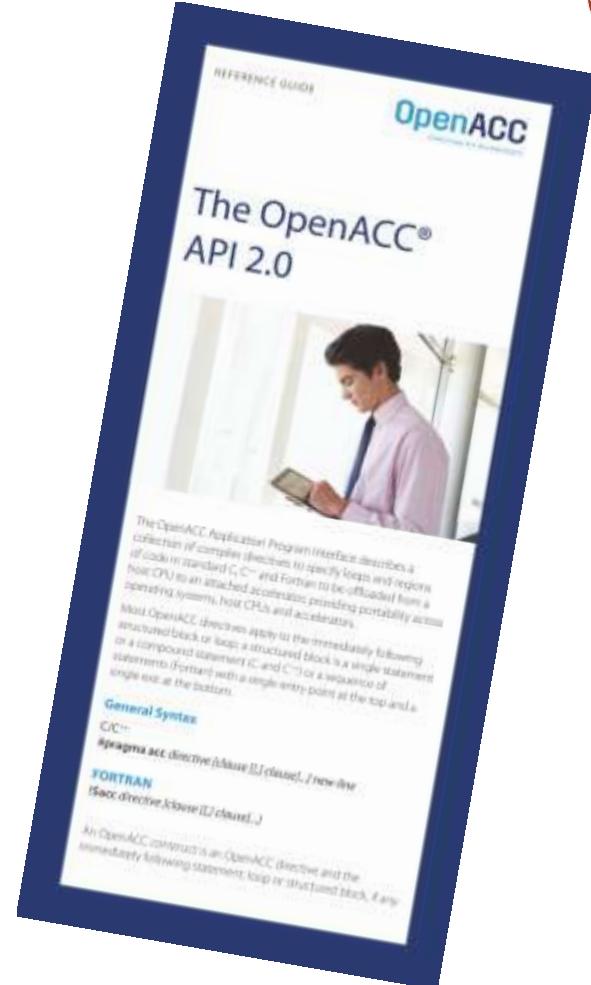


Contents

- **What is OpenACC?**
- **How does it work?**
- **What does it looks like?**
- **How do I use it?**
 - The basic directives
 - Enough to do the first two practicals
 - Advanced topics will follow in another lecture
- **Plus a few hints, tips, tricks and gotchas along the way**
 - Not all guaranteed to be relevant, useful (or even true)

- 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



- Compiler support: all now complete

- Cray CCE: complete OpenACC 2.0 in v8.2
- PGI Accelerator: version 12.6 onwards
- CAPS: Full support in v1.3
- gcc: work started in late 2013, aiming for 4.9
- Various other compilers in development

Execution and memory models

● Host-directed execution

- The main program executes on the host
 - The "host" can offload tasks to the "device" (accelerator, e.g. GPU)
 - tasks can be computation or data transfer between host and device
 - The host is responsible for managing the accelerator memory
 - as well as its own; allocating and freeing memory as needed
 - The host is responsible for synchronisation
 - ensuring offloaded tasks have completed

● Weak memory model

- The host and device have separate memories
 - There's no automatic data synchronisation going on in the background
- GPU memory is fragmented
 - there is no way to share data between all threads during computation
 - which leaves potential for race conditions

● The compiler and runtime help much more with OpenACC

COMPUTE | STORE | ANALYZE

Accelerator directives

- **Modify original source code with directives**

- Non-executable statements (comments, pragmas)
 - Can be ignored by non-accelerating compiler
 - CCE **-hnoacc** also suppresses 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 *** directive
 - Directives can be capitalized
- Continuation to extra lines allowed
 - **C/C++**: **** (at end of line to be continued)
 - **Fortran**:
 - Fixed form: **c\$acc&** or **!\$acc&** on continuation line
 - Free form: **&** at end of line to be continued
 - continuation lines can start with either **!\$acc** or **!\$acc&**

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

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

Conditional compilation

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

A first example

Execute a loop nest on the GPU

- Compiler does the work:

- Data movement
 - allocates/frees GPU memory at start/end of region
 - moves of data to/from GPU
- Loop schedule: spreading loop iterations over threads of GPU
 - OpenACC will "partition" (workshare) more than one loop in a loopnest
 - OpenMP only partitions the outer loop
- Caching (e.g. explicit use GPU shared memory for reused data)
 - automatic caching can be important
- Tune default behavior with optional clauses on directives

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

write-only

read-only

Accelerator kernels

- We call a loopnest that will execute on the GPU a "kernel"
 - this language is similar to CUDA
 - the loop iterations will be divided up and executed in parallel
- We have choice of two directives to create a kernel
 - parallel loop or kernels loop
 - both generate an accelerator kernel from a loopnest
 - the language is confusing
- Why are there two and what's the difference?
 - You can use either
 - We'll discuss the difference at the end of this lecture
 - This tutorial concentrates on using the parallel loop directive

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 GPU threads
 - First kernel initialises array
 - Compiler will determine a is write-only
 - Second kernel updates array
 - Compiler will determine a is read-write
 - Breaking parallel region=barrier
 - No barrier directive (global or within SM)

- Note:
 - Code can still be compiled for the CPU

Understanding data scoping

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

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

Data scoping in OpenACC (and OpenMP)

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

A more-explicit first version

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

- We could choose to make the data movements explicit
 - because we want to
 - or because the compiler is overcautious

- Note:
 - Array a is needlessly moved from/to GPU between kernels
 - You could call this "data sloshing"
 - This will have a big impact on performance

OpenACC data regions

- Data regions allow data to remain on the accelerator
 - e.g. for processing by multiple accelerator kernels
 - movement of specified arrays only occurs at start/end of region
 - these movements over-ride any automatic scoping
 - there is no automatic scoping: unspecified arrays don't move
 - Be careful: we have two unsynchronised copies of the specified arrays
 - Can contain host code, nested data regions and/or device kernels
 - In same routine: just omit OpenACC data clauses for these arrays
 - Separate routines: use the **present** clause instead of usual scoping
 - Structured data regions:
 - Fortran: **!\$acc data [data-clauses] ... !\$acc end data**
 - C/C++: **#pragma acc data [date-clauses] {...}**
 - Unstructured data regions (new in OpenACC v2):
 - Fortran: **!\$acc enter data [data-clauses] ... !\$acc exit data [data-clauses]**
 - C/C++: **#pragma acc enter data [clauses] ... #pragma acc exit data [clauses]**

A second version

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

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

- **No automatic synchronisation 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
```

- **present clause uses GPU version of b without data copy**
- Original calltree structure of program can be preserved
- One kernel now in subroutine (maybe in separate file)
 - OpenACC 1.0: function calls inside **parallel** regions required inlining
 - OpenACC 2.0: compilers will support nested parallelism

COMPUTE | STORE | ANALYZE

Reduction variables

- Reduction variables are a special case of private variables
 - where we will need to combine values across loop iterations
 - e.g. sum, max, min, logical-and etc. acting on a shared array
- We need to tell the compiler to treat this appropriately
 - Use the reduction clause for this (added to parallel loop directive)
 - same expression in OpenACC as in OpenMP
 - Examples:
 - sum: use clause reduction(+:t)
 - Note sum could involve adding and/or subtracting
 - max: use clause reduction(max:u)
- Note: OpenACC only allows reductions of scalars
 - not of array elements
 - advice:
 - try rewriting to use a temporary scalar in the loopnest for the reduction

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

Directives in summary

- **Compute regions**
 - created using **parallel loop** or **kernels loop** directives
- **Data regions**
 - created using **data** or **enter/exit data** directives
- **Data clauses are applied to:**
 - accelerated loopnests: **parallel** and **kernels** directives
 - here they over-ride relevant parts of the automatic compiler analysis
 - you can switch off all automatic scoping with **default(None)** clause (in v2)
 - data regions: **data** directive (plus **enter/exit data** in OpenACC v2)
 - Note there is no automatic scoping in data regions (arrays or scalars)
 - Shared clauses (**copy**, **copyin**, **copyout**, **create**)
 - supply list of scalars, arrays (or array sections)
 - Private clauses (**private**, **firstprivate**, **reduction**)
 - only apply to accelerated loopnests (**parallel** and **kernels** directives)
 - **present** clause (used for nested data/compute regions)

And take a breath...

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

Using the Cray Programming Environment

Alistair Hart

Cray Exascale Research Initiative Europe



Purpose of this talk

- **How to compile and run jobs**
 - You may know some of this already
 - daint (Cray XC30) and tödi (Cray XK7) have very similar environments
- **The practical exercises automate a lot of this**
 - The practicals are about learning OpenACC
 - not remembering system-specific details
 - The Makefiles, jobsheets etc. can be used as templates for your projects
- **It is still useful to have an overview of the system**

The system

- **Node types:**

- you log in and compile on a front end, login node
 - these nodes have no GPU, so you can't run jobs here
- jobs run on the compute nodes
 - each node has one CPU and one Nvidia Kepler K20X GPU
 - daint uses Intel Sandybridge CPUs; tödi uses AMD Interlagos CPUs

- **Running jobs**

- you run the jobs by submitting a jobscript to the SLURM batch system
 - compute jobs can't be run interactively from the front end command line

- **Filesystems:**

- there are two:
 - home directories; your is **\$HOME** and it is backed up
 - a fast, parallel lustre filesystem; your directory is **/scratch/todi/\$USER**
 - this is NOT backed up, and old files are periodically purged
- you should submit jobs from a directory on the lustre filesystem

Getting started

- Cray uses a linux-based environment on the login nodes
 - You will have a bash login shell by default
 - All the usual linux commands are available
 - Software versions are changed using Gnu modules (see **man module**)
 - To see which modules are currently loaded, type: **module list**
 - To see which modules are available, type: **module avail**
 - You can wildcard the end of the names, e.g.: **module avail PrgEnv***
 - You load a new module by typing: **module load <module name>**
 - Some modules (e.g. different compiler versions) conflict, so you should first "**module unload**" the old version (or use "**module swap**")

Programming Environments

- A number of different compilers are supported
 - You select these by loading a Programming Environment module
 - PrgEnv-cray for CCE (the default)
 - PrgEnv-pgi for PGI
 - PrgEnv-gnu for gcc, gfortran
 - Once one of these is loaded, you can then select a compiler version
 - CCE: `module avail cce`
 - PGI: `module avail pgi`
 - Gnu: `module avail gcc`
 - Swap to the most up to date version in each case
 - e.g. "`module avail cce`" to see the versions available
 - then "`module swap cce cce/<whatever>`"
 - For any GPU programming (CUDA, OpenCL, OpenACC...)
 - make sure you always: "`module load craype-accel-nvidia35`"
 - it is not always loaded by default

Using the compilers

- You use the compilers via wrapper functions
 - ftn for Fortran; cc for C; CC for C++
 - it doesn't matter which PrgEnv is loaded (same wrapper name)
- wrappers add optimisation, architecture-specific stuff and library paths
 - make sure you have the right one:
 - tödi: make sure module **craype-interlagos** is loaded
 - daint: make sure module **craype-sandybridge** is loaded
 - in many cases, you don't need any other compiler options
 - if you really want unoptimised code, you must use option **-O0**
- Further information
 - man pages for the wrapper commands give you general information
 - For more detail see the compiler-specific man pages
 - CCE: **crayftn**, **craycc**, **crayCC**
 - PGI: **pgfortran**, **pgcc**
 - GNU: **gfortran**, **gcc**
 - You will need the appropriate PrgEnv module loaded to see these

Some Cray Compilation Environment basics

- **CCE-specific features:**

- Optimisation: **-O2** is the default and you should usually use this
 - **-O3** activates more aggressive options; could be faster or slower
- **OpenMP** is supported by default.
 - if you don't want it, use **-hnoomp** compiler flag
- CCE only gives minimal information to stderr when compiling
 - to see more information, request a compiler listing file ("loopmark")
 - use flag **-hlist=a** for **ftn** and **cc**
 - writes a file with extension **.lst**
 - contains annotated source listing, followed by explanatory messages
 - each message is tagged with an identifier, e.g.: **ftn-6430**
 - to get more information on this, type: **explain <identifier>**
- For a full description of the Cray compilers, see the reference manuals at <http://docs.cray.com>.

Submitting jobs and the lustre filesystem

- You should submit jobs from the lustre filesystem
 - you can compile there as well if you wish
 - Directory `/scratch/todi/$USER` should be automatically created for you
 - you can create subdirectories if you want
- To submit a job, create a SLURM jobsript
 - there is a skeleton script provided as part of the tutorial materials
 - just rename the executable
 - note that command `aprun` is used within the jobsript to run the executable
 - submit the job using command: `sbatch <jobsript name>`
 - other options are specified in the jobsript header
 - a job number is returned
 - to view the queued and running jobs: `squeue`
 - to see just your jobs: `squeue -u $USER`
 - to stop a queued or running job: `scancel <job number>`

A sample SLURM script

- **SLURM parameters:**

- **nodes** is the total number of nodes required
- **ntasks** is the number of MPI ranks
- **ntasks-per-node** is the number of MPI ranks per node
 - Usually set to 1 for GPU jobs (unless you are using CUDA_PROXY)
- **cpus-per-task** is number of threads per rank
 - Usually set to 1 for GPU jobs
- See "**man sbatch**" for more details

- **Within the script:**

- If you are using OpenMP
 - you must manually set **OMP_NUM_THREADS** to **cpus-per-task** value
- Run job using **aprun** command; options usually match job parameters
 - **-n <ntasks>**
 - **-N <ntasks-per-node>**
 - **-d <cpus-per-task>**

A sample script

```
#!/bin/bash
#SBATCH --job-name="myjob"
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=00:01:00
#SBATCH --output="myjob.log"

# If you are using OpenMP, set the number of threads here
#export OMP_NUM_THREADS=<whatever>

# Now run the job with aprun

# As an example, we'll run the PGI pgaccelinfo program to give
# information about the GPU. To access this, we need to load
# PrgEnv-pgi:
module swap PrgEnv-cray PrgEnv-pgi
# In general, you don't need to do this.

# Run the command:
aprun -n1 -N1 -d1 pgaccelinfo
```

For information: Compiling CUDA

- **Compilation:**

- module load craype-accel-nvidia35
- Main CPU code compiled with PrgEnv "cc" wrapper
 - either PrgEnv-gnu for gcc; or PrgEnv-cray for craycc
- GPU CUDA-C kernels compiled with nvcc
 - nvcc -O3 -arch=sm_35
- PrgEnv "cc" wrapper used for linking
 - Only GPU flag needed: **-lcudart**
 - e.g. no CUDA **-L** flags needed (added in cc wrapper)

Starting the practicals

- Extract the tar file into your own directory and unpack
 - cp /users/ahart/Cray_OpenACC_training.tgz .
 - tar zxf Cray_OpenACC_training.tgz
 - cd Cray_OpenACC_training
- Read the LICENCE.txt file, then read the README.txt file
- Try running Practical0, to check it and your account work
 - type: source Tools/cray_setup.bash
 - this assumes you are running a bash shell
 - cd Practical0
 - bash submit.bash
 - squeue -u \$USER
 - to check if the job has finished. When it has...
 - cd /scratch/todi/\${USER}/<whatever>
 - directory name was reported by the submit.bash script
 - more *.log
 - then get coffee

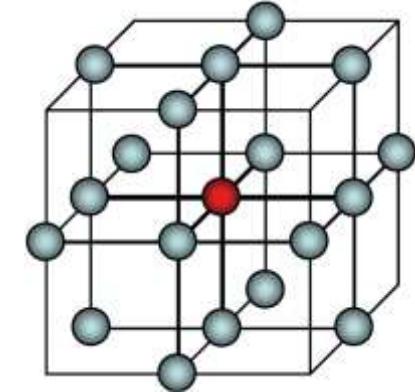
Porting a Simple Code

Worked example: scalar Himeno code



Practical 2

- These next slides give you a step-by-step guide to accelerating the Himeno code yourself
- The exact details are based on the Fortran version
 - line numbers, directive format
- But you can use the C version if you prefer
- You can compare to a set of reference code versions
 - see v01, v02 and v03
- Or you can just build and run these versions if you prefer
 - The results in these slides are from a Cray XK7



The Himeno Benchmark

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

Why use such a simple code?

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

The key questions in detail

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

First 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

Next stages to accelerating an application

4. Move up the calltree, adding higher-level data regions

- ideally, port entire application so data arrays live entirely on the GPU
- otherwise, minimise traffic between CPU and GPU
- This will give the single biggest performance gain

5. Only now think about performance tuning for kernels

- First correct any obviously inefficient scheduling on the GPU
 - This will give some good performance improvements
- Optionally, experiment with OpenACC tuning clauses
 - You may gain some final additional performance from this

• And remember Amdahl's law...

Run the initial CPU version of the code

- First we compile and run the code "as is"
 - source ../../Tools/cray_setup.bash
 - make clean
 - make VERSION=00 ACC=no
 - bash submit.bash himeno_v00.x
- When the job completes, we go to the run directory:
 - The location was reported by the submit.bash script; cd there
 - cat *.log
- Which includes the performance (in MFLOPS)
 - Measurement: Time (secs) : 4.8043761204462498
 - Measurement: Gosa : 0.137971540815963272E-02
 - Measurement: MFLOPS : 2853.7552548501367

Step 1: Himeno program structure

- **Code has two subprograms**

- **init_mt()** **initialises** the data array
 - Called once at the start of the program
- **jacobi()** performs iterative stencil **updates** of the data array
 - The number of updates is an argument to the subroutine and fixed
 - A summed residual is calculated, but not tested for convergence
 - This subroutine is called **twice**:
 - Each call is timed internally by the code
 - The first call does a **calibration run**
 - small, fixed number of iterations.
 - The time is used to estimate how many iterations could be done in one minute
 - The second call does a **measurement run**
 - computed number of iterations
 - The time is converted into a performance figure by the code
 - Actually, it is useful when testing to do a fixed number of iterations
 - Then we can use the value of the residual for a correctness check.
 - The next slide shows an edited version of the code
 - These slides discuss the Fortran version; there is also a C code

Step 1: Himeno program structure (contd)

calibration run

measurement run

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

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

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

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

  xmlops2 = flop*1.0d-6/cpu*nn

  PRINT *, ' Time: ',cpu
  PRINT *, ' Gosa: ',gosa
  PRINT *, ' MFLOPS: ',xmlops2
END PROGRAM himeno
```

- Now we look at the details of `jacobi()`

COMPUTE

Step 1: Structure of the jacobi routine

- **Iteration loop**

- must be sequential!

- **Apply stencil to p**

- create temporary **wrk2**
- residual **gosa** computed
 - details on the next slide

- **Update array p**

- from **wrk2**
- can be parallelised
- outer halo unchanged

```
SUBROUTINE jacobi(nn,gosa)
    iter_lp: DO loop = 1,nn
        ! compute stencil: wrk2, gosa from p
        <described on next slide>
        ! copy back wrk2 into p
        DO k = 2,kmax-1
            DO j = 2,jmax-1
                DO i = 2,imax-1
                    p(i,j,k) = wrk2(i,j,k)
                ENDDO
            ENDDO
        ENDDO
    ENDDO iter_lp
END SUBROUTINE jacobi
```

COMPUTE

STORE

ANALYZE

Step 1: The Jacobi computational kernel

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

```

gosa = 0d0
DO k = 2,kmax-1
  DO j = 2,jmax-1
    DO i = 2,imax-1
      S0=a(i,j,k,1)*p(i+1,j, k ) &
        +a(i,j,k,2)*p(i, j+1,k ) &
        +a(i,j,k,3)*p(i, j, k+1) &
        +b(i,j,k,1)*(p(i+1,j+1,k )-p(i+1,j-1,k ) &
          -p(i-1,j+1,k )+p(i-1,j-1,k )) &
        +b(i,j,k,2)*(p(i, j+1,k+1)-p(i, j-1,k+1) &
          -p(i, j+1,k-1)+p(i, j-1,k-1)) &
        +b(i,j,k,3)*(p(i+1,j, k+1)-p(i-1,j, k+1) &
          -p(i+1,j, k-1)+p(i-1,j, k-1)) &
        +c(i,j,k,1)*p(i-1,j, k ) &
        +c(i,j,k,2)*p(i, j-1,k ) &
        +c(i,j,k,3)*p(i, j, k-1) &
        + wrk1(i,j,k)

      ss = (s0*a(i,j,k,4)-p(i,j,k)) * bnd(i,j,k)
      gosa = gosa + ss*ss
      wrk2(i,j,k) = p(i,j,k) + omega*ss
    ENDDO
  ENDDO
ENDDO

```

COMPUTE

S1

Step 2: a first OpenACC kernel

- Start with most expensive
 - apply parallel loop
 - end parallel loop optional
 - advice: use it for clarity*
- reduction clause
 - as in OpenMP, not optional
- private clause
 - By default:
 - loop variables private
 - i, j, k
 - like OpenMP
 - scalar variables private
 - s0,ss
 - unlike OpenMP
 - so clause is optional here
 - Note that private arrays
 - always need private clause

```
gosa1 = 0d0

!$acc parallel loop reduction(+:gosa1)

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
```

Step 2: a first OpenACC kernel (contd)

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

```
gosa1 = 0d0

!$acc parallel loop reduction(+:gosa1) &
 !$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
```

Accelerate a first kernel

- Now we make a copy of the original file to OpenACC
 - cp himeno_F_v00.F90 himeno_F_v99.F90
- We accelerate the main computational kernel
 - Edit the following lines:
 - 292: !\$acc parallel loop reduction(+:gosa1)
 - 319: !\$acc end parallel loop
- Now build this code:
 - make VERSION=99 ACC=yes
- And examine the compiler loopmark commentary:
 - more himeno_F_v99.lst

Compiler feedback

- **Compiler feedback is extremely important**

- Did the compiler recognise the accelerator directives?
 - A good sanity check
- How will the compiler move data?
 - Only use data clauses if the compiler is over-cautious on the **copy***
 - Or you want to declare an array to be scratch space (**create** clause)
 - First major code optimisation: removing unnecessary data movements
- How will the compiler schedule loop iterations across GPU threads?
 - Did it parallelise the loop nests?
 - 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

Scheduling

G = accelerator kernel

```

292. + 1 G-----< !$acc parallel loop reduction(+:gosa1)
:
295. 1 G g-----< DO k = 2,kmax-1
296. + 1 G g 4-----< DO j = 2,jmax-1
297. 1 G g 4 g-----< DO i = 2,imax-1
298. 1 G g 4 g-----< S0 = a(i,j,k,1) * p(i+1,j ,k ) &
:
315. 1 G g 4 g-----> ENDDO
316. 1 G g 4-----> ENDDO
317. 1 G g-----> ENDDO
:
319. 1 G-----> !$acc end parallel loop

```

g = partitioned loop

Numbers denote
serial loops

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

ftn-6254 ftn: VECTOR File = himeno_F_v99.F90, Line = 284

A loop starting at line 284 was not vectorized because a recurrence was found on "wrk2" at line 292.

ftn-6405 ftn: ACCEL File = himeno_F_v99.F90, Line = 292

A region starting at line 292 and ending at line 319 was placed on the accelerator.

ftn-6430 ftn: ACCEL File = himeno_F_v99.F90, Line = 295

A loop starting at line 295 was partitioned across the thread blocks.

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

ftn-6509 ftn: ACCEL File = himeno_F_v99.F90, Line = 296

A loop starting at line 296 was not partitioned because a better candidate was found at line 297.

ftn-6412 ftn: ACCEL File = himeno_F_v99.F90, Line = 296

A loop starting at line 296 will be redundantly executed.

vectors, like **CUDA**:

i value(s) built from **threadidx.x**

ftn-6430 ftn: ACCEL File = himeno_F_v99.F90, Line = 297

A loop starting at line 297 was partitioned across the 128 threads within a threadblock.

128 threads within a threadblock.

Data movements

```

292. + 1 G-----< !$acc parallel loop reduction(:gosa1)
:
295.   1 G g-----<      DO k = 2,kmax-1
296. + 1 G g 4-----<      DO j = 2,jmax-1
297.   1 G g 4 g----<      DO i = 2,imax-1
298.   1 G g 4 g           S0 = a(i,j,k,1) * p(i+1,j ,k ) &
:
315.   1 G g 4 g---->      ENDDO
316.   1 G g 4----->      ENDDO
317.   1 G g----->      ENDDO
:
319.   1 G-----> !$acc end parallel loop

```

To learn more, use command:
explain ftn-6418

ftn-6418 ftn: ACCEL File = himeno_F_v99.F90, Line = 292
If not already present: allocate memory and copy whole array "p" to accelerator, free at line 319 (acc_copyin).

< same messages for: "a", "b", "c", "wrk1", "bnd" >

yes, as we expected

ftn-6416 ftn: ACCEL File = himeno_F_v99.F90, Line = 292
If not already present: allocate memory and copy whole array "wrk2" to accelerator, copy back at line 319 (acc_copy).

ftn-6415 ftn: ACCEL File = himeno_F_v99.F90, Line = 292

Allocate memory and copy variable "gosa1" to accelerator, copy back at line 319 (acc_copy).

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

Helping the compiler

- **The compiler is being too conservative**
 - It's worried that wrk2 has dependencies between loop iterations
- **We can help, but only where we need to**
 - Edit the following line:
 - 292: !\$acc parallel loop reduction(+:gosa1) copyout(wrk2)
- **Rebuild the code**
 - make VERSION=99 ACC=yes
- **Check the compiler commentary again**

Tuned data movements

```
292. + 1 G-----< !$acc parallel loop reduction(:gosa1) copyout(wrk2)
:
295.   1 G g-----<      DO k = 2,kmax-1
296. + 1 G g 4-----<      DO j = 2,jmax-1
297.   1 G g 4 g----<      DO i = 2,imax-1
298.   1 G g 4 g          S0 = a(i,j,k,1) * p(i+1,j ,k ) &
:
315.   1 G g 4 g---->      ENDDO
316.   1 G g 4----->      ENDDO
317.   1 G g----->      ENDDO
:
319.   1 G-----> !$acc end parallel loop
```

ftn-6419 ftn: ACCEL File = himeno_F_v99.F90, Line = 292

Allocate memory for whole array "wrk2" on accelerator, copy back at line 319 (acc_copyout).

ftn-6418 ftn: ACCEL File = himeno_F_v99.F90, Line = 292

If not already present: allocate memory and copy whole array "p" to accelerator, free at line 319 (acc_copyin).

< same messages for: "a", "b", "c", "wrk1", "bnd" >

ftn-6415 ftn: ACCEL File = himeno_F_v99.F90, Line = 292

Allocate memory and copy variable "gosa1" to accelerator, copy back at line 319 (acc_copy).

Run the first accelerated kernel

- Run this version
 - bash submit.bash himeno_v99.x
- When the job finishes, examine the log file
 - Measurement: Time (secs) : 10.121446384815499
 - Measurement: Gosa : 0.137971540815965701E-02
 - Measurement: MFLOPS : 1354.6002299205898
- First: the answer is still correct
 - Make sure you check the residual (**gosa**)
 - Note you won't get bitwise reproducibility between CPU and GPU
 - different compilers are also likely to give different results
- *Advice: make sure you have correctness checks*
 - e.g. checksums, residuals, representative array values
 - try to use double precision for checksums, residuals
 - even if the code is single precision
 - at least for the global sums or other reduction variables

Run the first accelerated kernel

- **Second: This first OpenACC version is slower.**
 - To find out why, we need more information.
- **First we look at the Cray runtime commentary**
 - In the run directory, edit the jobscrip: submit.wlm
 - uncomment line: export CRAY_ACC_DEBUG=2
 - Resubmit the job using the convenient script (written by submit.bash)
 - **bash resubmit.bash**
 - Wait for the the job to finish (check with squeue)
 - Then look in the replaced log file

CRAY_ACC_DEBUG=2

<For every iteration we see the following>

```
ACC: Start transfer 10 items from himeno_F_v99.F90:292
ACC:     allocate, copy to acc 'a' (68427792 bytes)
ACC:     allocate, copy to acc 'b' (51320844 bytes)
ACC:     allocate, copy to acc 'bnd' (17106948 bytes)
ACC:     allocate, copy to acc 'c' (51320844 bytes)
ACC:     allocate, copy to acc 'p' (17106948 bytes)
ACC:     allocate, copy to acc 'wrk1' (17106948 bytes)
ACC:     allocate 'wrk2' (17106948 bytes)
ACC:     allocate reusable, copy to acc <internal> (8 bytes)
ACC:     allocate reusable, copy to acc <internal> (4 bytes)
ACC:     allocate reusable <internal> (1008 bytes)
ACC: End transfer (to acc 222390336 bytes, to host 0 bytes)
ACC: Execute kernel jacobi_ckpt_L292_1 blocks:126 threads:128 async(auto) from himeno_F_v99.F90:292
ACC: Wait async(auto) from himeno_F_v99.F90:319
ACC: Start transfer 10 items from himeno_F_v99.F90:319
ACC:     free 'a' (68427792 bytes)
ACC:     free 'b' (51320844 bytes)
ACC:     free 'bnd' (17106948 bytes)
ACC:     free 'c' (51320844 bytes)
ACC:     free 'p' (17106948 bytes)
ACC:     free 'wrk1' (17106948 bytes)
ACC:     copy to host, free 'wrk2' (17106948 bytes)
ACC:     copy to host, done reusable <internal> (8 bytes)
ACC:     done reusable <internal> (4 bytes)
ACC:     done reusable <internal> (0 bytes)
ACC: End transfer (to acc 0 bytes, to host 17106956 bytes)
```

More profile information

- We can use the NVIDIA Compute Profiler to learn more:
 - Edit submit.wlm
 - Re-comment the line: `export CRAY_ACC_DEBUG=2`
 - Uncomment the line: `export COMPUTE_PROFILE=1`
 - Resubmit the job
- Examine the Compute Profiler output file
 - `more cuda_profile_0.log`

COMPUTE_PROFILE=1

<Header>

```
# CUDA_PROFILE_LOG_VERSION 2.0
# CUDA_DEVICE 0 Tesla K20X
# CUDA_CONTEXT 1
# TIMESTAMPFACTOR 1334047861ebec7b
method,gputime,cputime,occupancy
```

<Then at every iteration we see the following>

```
method=[ memcpyHtoD ] gputime=[ 27222.977 ] cputime=[ 27571.473 ]
method=[ memcpyHtoD ] gputime=[ 20372.352 ] cputime=[ 20636.215 ]
method=[ memcpyHtoD ] gputime=[ 6538.752 ] cputime=[ 6867.797 ]
method=[ memcpyHtoD ] gputime=[ 20333.984 ] cputime=[ 20594.152 ]
method=[ memcpyHtoD ] gputime=[ 6538.912 ] cputime=[ 6867.491 ]
method=[ memcpyHtoD ] gputime=[ 6547.712 ] cputime=[ 6871.381 ]
method=[ memcpyHtoD ] gputime=[ 1.632 ] cputime=[ 8.488 ]
method=[ memcpyHtoD ] gputime=[ 0.928 ] cputime=[ 6.142 ]
method=[ jacobi_ckpt_L292_1 ] gputime=[ 2079.296 ] cputime=[ 25.261 ] occupancy=[ 0.312 ]
method=[ memcpyDtoH ] gputime=[ 35016.480 ] cputime=[ 37985.234 ]
method=[ memcpyDtoH ] gputime=[ 2.464 ] cputime=[ 19.153 ]
```

Profiling with CrayPAT

- We really need an application-wide view using CrayPAT
 - rather than an event-by-event account
- Back in the source directory:
 - make clean
 - module load perftools
 - make VERSION=99 ACC=yes
 - pat_build -w himeno_v99.x
 - Option -w does tracing
 - -u would include tracing of CPU routines as well
- Then we run the instrumented code
 - bash submit.bash himeno_v99.x+pat
- When it completes, in the run directory
 - pat_report *.xf

pat_build -w

Table 1: Profile by Function Group and Function

Time%	Time	Imb.	Imb.	Calls	Group
		Time	Time%		Function
100.0%	10.801079	--	--	917.0	Total
100.0%	10.800891	--	--	516.0	USER
86.0%	9.293427	--	--	103.0	jacobi_.ACC_COPY@li.292
7.8%	0.838496	--	--	1.0	main_
4.2%	0.448701	--	--	103.0	jacobi_.ACC_COPY@li.319
2.0%	0.215345	--	--	103.0	jacobi_.ACC_SYNC_WAIT@li.319

Table 2: Time and Bytes Transferred for Accelerator Regions

Host Time%	Host Time	Acc Time	Acc Copy In (MBytes)	Acc Copy Out (MBytes)	Events	Calltree
100.0%	9.963	9.935	21845	1680	515	Total
100.0%	9.963	9.935	21845	1680	515	main_
3						jacobi_
4	93.3%	9.293	9.275	21845	--	jacobi_.ACC_REGION@li.292
4	4.5%	0.449	0.444	--	1680	jacobi_.ACC_COPY@li.319
4	2.2%	0.215	--	--	--	jacobi_.ACC_SYNC_WAIT@li.319

Step 3: Optimising data movements

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

Step 3: Structure of the jacobi routine

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

```
SUBROUTINE jacobi(nn,gosa)

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

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

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

    ENDDO iter_lp
    !$acc end data

END SUBROUTINE jacobi
```

COMPUTE

STORE

ANALYZE

Reducing data movements

- **To reduce data movements we add a data region**
 - We put it in jacobi, outside the iteration loop:
 - 280: !\$acc data copy(p) copyin(a,b,c,bnd,wrk1) create(wrk2)
 - 340: !\$acc end data
 - We need to remove the copyout clause for wrk2:
 - 292: !\$acc parallel loop reduction(+:gosa1)
 - We also need to accelerate the other loopnest:
 - 321: !\$acc parallel loop
 - 331: !\$acc end parallel loop
- **Now we build and submit this version**
 - module unload perftools
 - make clean
 - make VERSION=99 ACC=yes
 - bash submit.bash himeno_v99.x

Improved performance

- When the job completes, examine the log file:
 - Measurement: Time (secs) : 0.33722697710618377
 - Measurement: Gosa : 0.137971540815965701E-02
 - Measurement: MFLOPS : 40656.6334569459
- The code is much faster.
 - To see why, uncomment the CRAY_ACC_DEBUG line in submit.wlm
 - And resubmit the job
 - When it completes, examine the replaced log file

CRAY_ACC_DEBUG=2

<Start of data region>

```
ACC: Start transfer 7 items from himeno_F_v99.F90:280
ACC:     allocate, copy to acc 'a' (68427792 bytes)
ACC:     allocate, copy to acc 'b' (51320844 bytes)
ACC:     allocate, copy to acc 'bnd' (17106948 bytes)
ACC:     allocate, copy to acc 'c' (51320844 bytes)
ACC:     allocate, copy to acc 'p' (17106948 bytes)
ACC:     allocate, copy to acc 'wrk1' (17106948 bytes)
ACC:     allocate 'wrk2' (17106948 bytes)
ACC: End transfer (to acc 222390324 bytes, to host 0 bytes)
```

<For each loop iteration... (see next slide)>

<After iteration loop finishes, close of data region>

```
ACC: Wait async(auto) from himeno_F_v99.F90:340
ACC: Start transfer 7 items from himeno_F_v99.F90:340
ACC:     free 'a' (68427792 bytes)
ACC:     free 'b' (51320844 bytes)
ACC:     free 'bnd' (17106948 bytes)
ACC:     free 'c' (51320844 bytes)
ACC:     copy to host, free 'p' (17106948 bytes)
ACC:     free 'wrk1' (17106948 bytes)
ACC:     free 'wrk2' (17106948 bytes)
ACC: End transfer (to acc 0 bytes, to host 17106948 bytes)
```

CRAY_ACC_DEBUG=2 continued

<Start of data region... (see previous slide)>

<For each loop iteration>

```
ACC: Start transfer 3 items from himeno_F_v99.F90:293
ACC:     allocate reusable, copy to acc <internal> (8 bytes)
ACC:     allocate reusable, copy to acc <internal> (4 bytes)
ACC:     allocate reusable <internal> (1008 bytes)
ACC: End transfer (to acc 12 bytes, to host 0 bytes)
ACC: Execute kernel jacobi_ckpt_L293_1 blocks:126 threads:128 async(auto) from himeno_F_v99.F90:293
ACC: Wait async(auto) from himeno_F_v99.F90:320
ACC: Start transfer 3 items from himeno_F_v99.F90:320
ACC:     copy to host, done reusable <internal> (8 bytes)
ACC:     done reusable <internal> (4 bytes)
ACC:     done reusable <internal> (0 bytes)
ACC: End transfer (to acc 0 bytes, to host 8 bytes)
ACC: Execute kernel jacobi_ckpt_L322_2 blocks:126 threads:128 async(auto) from himeno_F_v99.F90:322
```

<After iteration loop finishes, close of data region... (see previous slide)>

Profiling this version

- We can generate a CrayPAT profile as before:
 - make clean
 - module load perftools
 - make VERSION=99 ACC=yes
 - pat_build -w himeno_v99.x
 - bash submit.bash himeno_v99.x+pat
- When the job completes, we generate a profile
 - pat_report *.xf

pat_build -w

Table 1: Profile by Function Group and Function

Time%	Time	Imb.	Imb.	Calls	Group
		Time	Time%		Function
100.0%	0.755941	--	--	1131.0	Total
<hr/>					
100.0%	0.755749	--	--	730.0	USER
<hr/>					
40.9%	0.308847	--	--	1.0	main_
28.4%	0.214851	--	--	103.0	jacobi_.ACC_SYNC_WAIT@li.320
24.1%	0.182321	--	--	2.0	jacobi_.ACC_COPY@li.280
4.3%	0.032620	--	--	103.0	jacobi_.ACC_COPY@li.293
1.0%	0.007862	--	--	2.0	jacobi_.ACC_COPY@li.340
<hr/>					

Table 2: Time and Bytes Transferred for Accelerator Regions (see next slide)

pat_build -w

Table 1: Profile by Function Group and Function (see previous slide)

Table 2: Time and Bytes Transferred for Accelerator Regions

Host Time%	Host Time	Acc Time	Acc Copy In (MBytes)	Acc Copy Out (MBytes)	Events	Calltree
100.0%	0.447	0.436	424.177	32.630	729	Total
100.0%	0.447	0.436	424.177	32.630	729	main_
3						jacobi_
						jacobi_.ACC_DATA_REGION@li.280
4	56.8%	0.254	0.217	0.001	0.001	515 jacobi_.ACC_REGION@li.293
5	48.1%	0.215	--	--	--	103 jacobi_.ACC_SYNC_WAIT@li.320
5	7.3%	0.033	0.001	0.001	--	103 jacobi_.ACC_COPY@li.293
4	40.8%	0.182	0.180	424.176	--	2 jacobi_.ACC_COPY@li.280
4	1.8%	0.008	0.008	--	32.629	2 jacobi_.ACC_COPY@li.340

A synchronous profile

- **GPU kernels launch asynchronously**
 - So the compute time shows up in the SYNC_WAIT events
- **We can switch off the automatic asynchronicity**
 - This can give a clearer profile, but it may be skewed
 - We use CCE flag `-hacc_model=auto_async_none`
- **Back in the source directory**
 - `make clean`
 - `make VERSION=99 ACC=yes ACC_MODEL=auto_async_none`
 - `pat_build -w himeno_v99.x`
 - `bash submit.bash himeno_v99.x+pat`
- **When the job completes**
 - `pat_report *.xf`

pat_build -w with auto_async_none

Table 1: Profile by Function Group and Function

Time%	Time	Imb.	Imb.	Calls	Group
		Time	Time%		Function
100.0%	0.756016	--	--	1026.0	Total
100.0%	0.755826	--	--	625.0	USER
40.7%	0.307802	--	--	1.0	main_
28.7%	0.216719	--	--	103.0	jacobi_.ACC_KERNEL@li.293
24.1%	0.182522	--	--	2.0	jacobi_.ACC_COPY@li.280
4.4%	0.033479	--	--	103.0	jacobi_.ACC_KERNEL@li.322
1.1%	0.008124	--	--	2.0	jacobi_.ACC_COPY@li.340

Table 2: Time and Bytes Transferred for Accelerator Regions (see next slide)

pat_build -w with auto_async_none

Table 1: Profile by Function Group and Function (see previous slide)

Table 2: Time and Bytes Transferred for Accelerator Regions

Host Time%	Host Time	Acc Time	Acc Copy In (MBytes)	Acc Copy Out (MBytes)	Events	Calltree
100.0%	0.448	0.437	424.177	32.630	624	Total
100.0%	0.448	0.437	424.177	32.630	624	main_
3						jacobi_
4	49.9%	0.224	0.217	0.001	0.001	jacobi_.ACC_REGION@li.293
5	48.4%	0.217	0.214	--	--	jacobi_.ACC_KERNEL@li.293
4	40.7%	0.183	0.180	424.176	--	2 jacobi_.ACC_COPY@li.280
4	7.5%	0.034	0.031	--	--	206 jacobi_.ACC_REGION@li.322
5	7.5%	0.034	0.031	--	--	103 jacobi_.ACC_KERNEL@li.322
4	1.8%	0.008	0.008	--	32.629	2 jacobi_.ACC_COPY@li.340

Step 4: Further optimising data movements

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

Adding a data region

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

COMPUTE

```

PROGRAM himeno
  CALL initmt

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

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

END PROGRAM himeno

```

```
SUBROUTINE jacobi(nn,gosa)
```

```

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

    ENDDO iter_lp
!$acc end data

END SUBROUTINE jacobi

```

Step 4: Going further

- **Best solution is to port entire application to GPU**
 - data regions span entire use of arrays
 - all enclosed loopnests accelerated with OpenACC
 - no significant data transfers
- **Expand outer data region to include call to initialisation routine**
 - arrays can now all be declared as scratch space with `create` clause
 - need to accelerated loopnests in `initmt()`, declaring arrays `present`
- **N.B. No easy way to ONLY allocate arrays in GPU memory**
 - CPU version is now dead space, but
 - GPU memory is usually the limiting factor, so usually not a problem
 - Can use OpenACC API calls to do this, if really want to
 - Means more OpenACC-specific code changes

Porting entire application

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

```
PROGRAM himeno

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

    CALL jacobi(3,gosa) ! plus timing calls
    CALL jacobi(nn,gosa) ! plus timing calls

!$acc end data

END PROGRAM himeno
```

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

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

END SUBROUTINE initmt
```

Adding an outer data region

- **We can reduce data movement s further**

- with a data region in the main program
- Edit the following lines:
 - 116: !\$acc data create(a,b,c,p,wrk1,bnd,wrk2)
 - 198: !\$acc end data
- Change the data region in jacobi to use present clause:
 - 280: !\$acc data present(p,a,b,c,bnd,wrk1,wrk2)
- We also need to accelerate the initmt loopnests:
 - 212: !\$acc data present(a,b,c,p,wrk1,bnd)
 - 261: !\$acc end data
 - 214: !\$acc parallel loop
 - 236: !\$acc end parallel loop
 - 237: !\$acc parallel loop
 - 259: !\$acc end parallel loop

Add outer data region

- Build and run the new code
 - module unload perftools
 - make clean
 - make VERSION=99 ACC=yes
 - bash submit.bash himeno_v99.x
- Then examine the performance in the log file:
 - Measurement: Time (secs) : 0.24414781527593732
 - Measurement: Gosa : 0.137971540815965701E-02
 - Measurement: MFLOPS : 56156.609816492906

Profiling the final version

- We can profile this version of the code
 - using -hacc_model=auto_async_none
- Back in the source directory
 - module load perftools
 - make clean
 - make VERSION=99 ACC=yes ACC_MODEL=auto_async_none
 - pat_build -w himeno_v99.x
 - bash submit.bash himeno_v99.x+pat
- When the job completes, we process the tracing file
 - pat_report *.xf

pat_build -w , with auto_async_none

Table 1: Profile by Function Group and Function

Time%	Time	Imb.	Imb.	Calls	Group
		Time	Time%		Function
100.0%	0.263683	--	--	1140.0	Total
99.9%	0.263520	--	--	739.0	USER
81.8%	0.215700	--	--	103.0	jacobi_.ACC_SYNC_WAIT@li.320
13.9%	0.036764	--	--	103.0	jacobi_.ACC_COPY@li.293
1.4%	0.003685	--	--	103.0	jacobi_.ACC_COPY@li.320

Table 2: Time and Bytes Transferred for Accelerator Regions (see next slide)

pat_build -w , with auto_async_none

Table 1: Profile by Function Group and Function (see previous slide)

Table 2: Time and Bytes Transferred for Accelerator Regions

Host Time%	Host Time	Acc Time	Acc Copy In (MBytes)	Acc Copy Out (MBytes)	Events	Calltree
100.0%	0.264	0.255	0.001	0.001	738	Total
100.0%	0.264	0.255	0.001	0.001	738	main_main_.ACC_DATA_REGION@li.116
3 98.9%	0.261	0.249	0.001	0.001	727	jacobi_jacobi_.ACC_DATA_REGION@li.280
5 98.1%	0.259	0.218	0.001	0.001	515	jacobi_.ACC_REGION@li.293
6 81.8%	0.216	--	--	--	103	jacobi_.ACC_SYNC_WAIT@li.320
6 14.0%	0.037	0.001	0.001	--	103	jacobi_.ACC_COPY@li.293
6 1.4%	0.004	0.002	--	0.001	103	jacobi_.ACC_COPY@li.320

Elimination of data transfers

- The remaining data transfers are small compiler internals
 - associated with the reduction variables
 - and you have no control over these
 - so you shouldn't worry about them
 - if these really are the limiting factor in your application, well done!
- To check this
 - Run with CRAY_ACC_DEBUG=2
 - Use the same procedure as before
 - Then look at the logfile

OpenACC and/or OpenMP (more advanced)

- So far all performance comparisons are unfair
 - full accelerator versus single CPU core
- Need an OpenMP version to fully exercise CPU
 - cd back to source directory
 - module unload perftools
 - module unload craype-accel-nvidia35
 - make clean
 - make VERSION=99 OMP=yes ACC=no
 - bash submit.bash himeno_v99.x N
 - where N is the number of OpenMP threads
 - try different values (up to 8) and see which performs best
- Overall GPU speedup: 3-5x
 - This is the sort of figure we expect
 - Kepler GPU memory bandwidth around 5x compared to Sandybridge CPU

In summary

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

In summary... continued

- **Further performance tuning**

- data region gave a **20-40x** speedup; kernel tuning is secondary
- Low-level languages like **CUDA**
 - offer more direct control of the hardware
 - but **OpenACC** is much easier to use
 - should get close to **CUDA** performance
- Remember Amdahl's Law:
 - speed up the compute of a parallel application,
 - and soon become network bound
 - Don't waste time trying to get an extra 10% in the compute
 - You are better concentrating your efforts on tuning the comms or I/O

- **Bottom line:**

- **3-5x** speedup from **7** directive pairs in **200** lines of Fortran/C
- performance compares GPU to the complete CPU

The OpenACC programming model:

2. some extra details

Alistair Hart

Cray Exascale Research Initiative Europe



Contents

- This lecture collects together some extra useful concepts:
 - the `update` directive
 - array sectioning and unshaped pointer errors
 - using accelerator-resident data: `present_or_*` clauses
 - the `if` clause
 - `parallel` vs kernels
 - `parallel loop` vs. `parallel` and `loop`
 - race conditions
 - a few "gotchas"

Minimising data movements

- This is the single-biggest OpenACC optimisation for GPUs
- There are three techniques:
 - Keep data on the GPU as long as possible
 - use data regions and port all enclosed loopnests (as we have seen)
 - Only move arrays when you need to
 - using the `update` directive
 - Only move the data you need to move
 - using array sections

The update directive

- **Data regions keep data on device**
 - can span multiple compute kernels and serial (host) code
 - create copies of data arrays on device for duration of data region
 - host, device copies only synchronised at start/end of data region
 - as requested by explicit data clauses
- **You can synchronise copies manually within a data region**
 - for instance:
 - to copy a halo buffer back to the host for communication
 - to copy values of an array to the CPU for checking or printing
- **You do this using the update directive, for instance:**
 - **update host(a,b)** copies entire arrays a,b from device to host
 - OpenACC 2.0: can use **self** instead of **host**
 - **update device(a,b)** copies entire arrays a,b from host to device

Array sections

- Sometimes we only need to move part of an array
 - array section notation allows this, using ":" notation
 - syntax differs slightly between languages
 - Fortran uses **start:end**, so first N elements is **a(1:N)**
 - C/C++ uses **start:length**, so first N elements is **b[0:N]**
 - **Advice:** be careful when switching languages!
 - Use profiler, CRAY_ACC_DEBUG commentary to see how much data moved
- Sections allowed in **data clauses** and with **update**
- For multi-dimensional arrays
 - specified sections must be a contiguous block of memory
 - can only specify one incomplete section on slowest-moving index
 - Fortran: slowest index is right-most
 - so **a(1:N,2:N-1)** is allowed, but **a(2:N-1,1:N)** is not
 - C/C++: slowest index is left-most
 - so **b[1:N-2][0:N]** is allowed, but **b[0:N][1:N-2]** is not

Unshaped pointers

- "Unshaped pointer" compiler/runtime errors
 - Fortran arrays have a complicated data structure
 - includes a descriptor that contains information about size and shape
 - the compiler therefore knows how much data to transfer
 - C/C++: arrays are often just pointers
 - especially if the arrays were dynamically allocated or passed by reference
 - How many bytes should be transferred here: `copy(c)` ?
 - So you usually need to be more explicit
 - You need to put in data clauses
 - And specify the slicing (even if this is the whole array): `copy(c[0:N])`

Sharing data between kernels

- **If you are using a data region around kernels**
 - Must ensure that the runtime uses the shared data already present
- **If the kernel is in the same routine as the data region**
 - just don't mention those bits of data in the **parallel/kernels** clauses
- **If the kernel is in a different routine to the data region**
 - On the **parallel** or **kernels** directive:
 - Specify the relevant data with **present** clause
 - instead of other shared clauses (**copy**, **copyin**, **copyout**, **create**)
 - don't rely on automatic scoping for shared data in this case
- **If an array is declared **present**, but is not on accelerator**
 - you get a runtime error and the program crashes
 - This is usually what you would like to happen
 - rather than running to completion with the wrong answer
 - But there are other ways to do things...

COMPUTE | STORE | ANALYZE

Use-case one

- A routine should **always** process data on the device
 - but it is called in two different ways within the code
 - sometimes from inside a data region, when data is **present**
 - sometimes from outside a data region, when data should be copied
- To do this, instead of using **present** clause
 - on **parallel** or **kernels** directives, instead of **present** clause, use:
 - **present_or_copy[in|out]**, **present_or_create**
 - or the short form: replace **present_or_** by **p**, e.g. **pcopyin**, **pcreate**
 - The runtime will check if data is already on the device
 - if so, it uses that version (with no data copying)
 - if not, it does the prescribed data copying either side of the kernel
 - Advice: only use **present_or_*** versions if you really must
 - most "real" codes have a simple calltree:
 - data to be processed on the accelerator should always be present
 - "not present" runtime errors are a useful debugging tool

Use-case two

- A routine is called in two different ways within the code
 - sometimes from inside a data region, when relevant data is on the host
 - sometimes from outside a data region, when the data is on the device
- In each case, the data should be processed "locally"
- if clause: applied to parallel or kernels directives
 - decide at runtime whether to execute loopnest on CPU or accelerator
 - executes on GPU if .TRUE. (Fortran) or 1 (C/C++), otherwise on CPU
- But what is the argument to the if clause?
 - either we pass a separate LOGICAL to the routine
 - or we can use the runtime API to check if data is present
 - this is an advanced topic

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 loop is really a composite directive**

- You can use them separately if you prefer
 - data clauses go on the **parallel** directive
 - a **loop** directive must follow the **parallel** directive
 - Fortran: **end loop** directives are optional
- You can put extra **loop** directives above other loops in the loopnest
 - There is no reason to always do this:
 - OpenACC automatically partitions loops within the loopnest
 - Extra **loop** directives are useful when we are tuning the loop schedule
 - We'll come to this later

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

Inside a parallel region

- At the start we launch all the threads
 - executing as a set of threadblocks, spread over the SMs
- The threads execute the code in the parallel region
- There are two modes of execution
 - redundantly: identically for every thread
 - or possibly just by one thread (it's implementation-dependent)
 - partitioned: with the work shared between threads
- A parallel region can contain:
 - scalar code
 - This would execute redundantly
 - a partitioned loopnest
 - This will execute in a partitioned manner
 - an unpartitioned loopnest (no loop directive)
 - This would also execute redundantly

COMPUTE | STORE | ANALYZE

parallel region race conditions

- A **parallel region can span multiple code blocks**
 - Can be a mixture of scalar code, partitioned and unpartitioned loops
- **There is an extreme risk of race conditions if you do this**
 - 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
 - There is no synchronisation between code blocks
 - because no synchronisation between threadblocks

```
!$acc parallel
t = 0
u = 0

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

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

parallel and loop for overlap?

"But multiple code blocks might give me overlap"

- **My advice: don't...**

- GPU threads are very lightweight (unlike OpenMP)
 - so don't worry about having extra parallel regions
- the **async** clause is a better way to get overlap

- **... but if you feel you must**

- begin with separate **parallel loop** directives
- make sure the code is correct
- only then separate directives to tune performance
 - making sure the kernels are independent and no race conditions

- **This is similar to using OpenMP on the CPU**

- If you have multiple do/for directives inside **omp parallel** region
- only introduce **nowait** clause when you are sure the code is working
 - and watch out for race conditions

parallel region gotchas

- **No loop directive**

- The code will (or may) run redundantly
 - Every thread does every loop iteration
 - Not usually what we want

- **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.

- **Multiple kernels**

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

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

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

```
!$acc parallel
 !$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
```

Extra loop directives and reductions

- If you put in extra loop directives...
 - e.g. to add tuning clauses
- ... you must make sure you repeat the reduction clause
 - it should appear on every explicit loop directive
 - that contains the reduction
- It is easy to forget this
 - The code will compile and execute, but give incorrect results

```
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
```

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

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

```
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
```

```
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!

COMPUTE | STORE | ANALYZE

Correct!

OpenACC and/or OpenMP (more advanced)

- A lot of similarities between OpenACC and OpenMP
 - We mean OpenMP for the CPU here
- But also some important differences
- OpenMP and OpenACC can coexist in same source code
 1. As alternatives for compilation on a given loopnest
 - Use compiler flags to enable/disable: -hnoomp, -hnoacc for CCE
 2. On separate code regions
 3. You can nest OpenACC inside OpenMP (but not other way round)
 - Cray OpenACC runtime is OpenMP-threadsafe
 - But beware of too much data movement
- If you want some routines to be always OpenMP, but others to be sometimes OpenMP and sometimes OpenACC
 - you'll need to preprocess using the _OPENMP and _OPENACC macros

OpenMP then OpenACC

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

Aside: OpenACC and OpenMP (naïve)

```

!$acc data etc.

iteration: DO loop = 1, nn

gosa = 0d0

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

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

gosa = gosa + gosal

ENDDO iteration
!$acc end data

```

```

iteration: DO loop = 1, nn

gosa = 0d0

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

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

gosa = gosa + gosal

ENDDO iteration

```



Aside: OpenACC and OpenMP (better)

```
!$acc data etc.
```

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

```
gosa = 0d0
```

```
gosal = 0d0
```

```
!$acc parallel loop
```

```
reduction(+:gosal) etc.
```

```
<stencil loopnest>
```

```
!$acc end parallel loop
```

```
!$acc parallel loop
```

```
<copy loopnest>
```

```
!$acc end parallel loop
```

```
gosa = gosa + gosal
```

```
ENDDO iteration
```

```
!$acc end data
```

```
!$omp parallel
```

```
private(s0,ss,gosal) etc.
```

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

```
!$omp barrier
```

```
!$omp master
```

```
gosa = 0d0
```

```
!$omp end master
```

```
gosal = 0d0
```

```
!$omp do
```

```
<stencil loopnest>
```

```
!$omp end do
```

```
!$omp do
```

```
<copy loopnest>
```

```
!$omp end do nowait
```

```
!$omp critical
```

```
gosa = gosa + gosal
```

```
!$omp end critical
```

```
ENDDO iteration
```

```
!$omp end parallel
```

COMPUTE

STORE

ANALYZE

Aside: OpenACC versus OpenMP

- **One OpenMP threadteam for whole iteration loop**
 - OpenMP threads are "heavyweight" (expensive to create teams)
 - need **barrier/master/critical** regions for synchronisation
 - **gosa1** is a thread-private variable; **critical** region used for reduction
- **Multiple OpenACC parallel loop kernels**
 - GPU kernels execute as threadblocks on different SMs
 - need separate kernels to synchronise
 - no global synchronisation method within a kernel
 - GPU threads are "lightweight" (kernels cheap to launch)
 - **gosa1** is a shared reduction loop variable
- **Can we use a single OpenACC parallel region?**
 - Containing scalar code and multiple **loop** nests?
 - Analogous to version of code with multiple OpenMP **parallel** region
 - **NO!** Race conditions between kernels and with scalar code
 - think of **acc end loop** as comparable to **omp end do nowait**

OpenACC 3: Performance Tuning

Alistair Hart
Cray Exascale Research Initiative Europe



COMPUTE

STORE

ANALYZE

Contents

- How kernels execute on a GPU
- Scheduling
 - the importance of vectorisation
 - the `vector_length` clause
 - tuning the schedule: `collapse` and `worker` clauses
 - further tuning: `cache` and `tile`

The next step

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

How a kernel is executed on a GPU

- A GPU kernel is executed by a large pool of threads
 - Whether we use OpenACC or CUDA
 - The threads are logically divided into sets called **threadblocks**
 - Each **threadblock** executes on a different piece of hardware
 - Symmetric Multiprocessor (SM)
 - (Almost) like a single vector processor
- Within a **threadblock**
 - The threads are logically divided into sets called **warps**
 - threads within a **warp** (32 threads) are executed concurrently
 - (almost) like a set of vector instructions, each of fixed width 32
 - so the **threadblock** executes as a sequence of concurrent **warps**
- The CUDA programming model
 - does not make **warps** explicit
 - but you need to know about them to understand code performance

OpenACC scheduling

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

Kernel vectorisation

- After minimising data movements, next optimisation:
 - make sure all the kernels **vectorise**
 - meaning the compiler is efficiently using vector instructions on GPU
- How can I tell if there is a problem?
 - if a kernel is surprisingly slow on accelerator
 - in a very-different place in the profile compared to running on CPU
 - then examine the compiler commentary
 - CCE: compile with flag **-hlist=a** to generate ***.lst** loopmark file
- Should see loop iterations divided **both**:
 - over threads within a threadblock (**vector**) **and**
 - over threadblocks (**gang**)

Vectorisation and loopmark

- With CCE, you should see:

- For a single loop:
 - should be divided over both levels of parallelism
 - look for this in the loopmark: Gg
- For a loopnest:
 - Two (or maybe three) loops should be divided:
 - look for G and 2 (or maybe 3) g-s
 - possibly with some numbers in between

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

A loop starting at line 172 was partitioned across the thread blocks.

A loop starting at line 174 was partitioned across the 128 threads within a threadblock.

COMPUTE

STORE

ANALYZE

Kernel optimisation

- **Making sure all the kernels vectorise**
 - should have one loop "partitioned across threads within a threadblock"
- **Which loop do we want to vectorise?**
 - generally want to vectorise the innermost loop
 - usually labels fastest-moving array index, for coalescing
- **If it is not vectorised, can we make it vectorise?**
 - Can loop iterations be computed in any order?
 - No? Then can we rewrite code
 - avoid loop-carried dependencies
 - e.g. buffer packing: calculate rather than increment
 - these rewrites will probably perform better on CPU
 - there is a lot of literature on vectorisable algorithms

Replace:

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

By:

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

Forcing compiler to vectorise

- **Can loop iterations be computed in any order?**
 - Yes? Then we should guide the compiler
 - Either with a gentle hint:
 - put "acc loop independent" directive above this loop
 - recompile the routine and check compiler feedback to see if this worked
 - Or with a direct order:
 - put "acc loop vector" directive above this loop
 - check the code is still correct (as well as running faster)
 - the compiler might not be vectorising the loop for a good reason
- **Advice:**
 - Remember to repeat any **reduction** clauses on new **loop** directives

Changing the vectorisation with seq

- If the inner loop is vectorising but performance is still bad

- Is the inner loop really the one to vectorise in this case?

- in this example, we should vectorise the **i**-loop
 - because we happen to know **mmax** is small here
 - or because CrayPAT loop-level profiling told us this

- Put "acc loop seq" directive above **m**-loop
 - then executed "redundantly" by every thread
 - also **t** is now an **i**-loop private scalar
 - rather than a reduction variable
 - this should also help performance

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

- The compiler should now vectorise a different loop

- usually the next one up in the loopnest
 - in this case there is no choice but the **i**-loop
- if it is not the next one up in the loopnest
 - you will need to put an "acc loop vector" directive in the right place

An aside on this example

- We forced vectorisation of the **i**-loop
 - because **mmax** was too small
 - small loops will not give the GPU enough work
- Performance is still likely to be bad
 - a warp is 32 threads, executing as a vector
 - each thread has a different **i**-value
 - data for the warp is done in vector loads
 - but **c(m,1:32)** is not a contiguous chunk of memory
 - so we will need multiple vector loads for each warp
 - loads/stores from global memory are slow, so performance will suffer
 - this is a hardware feature, not a limitation of OpenACC

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

More aside on this example

- So what can we do?

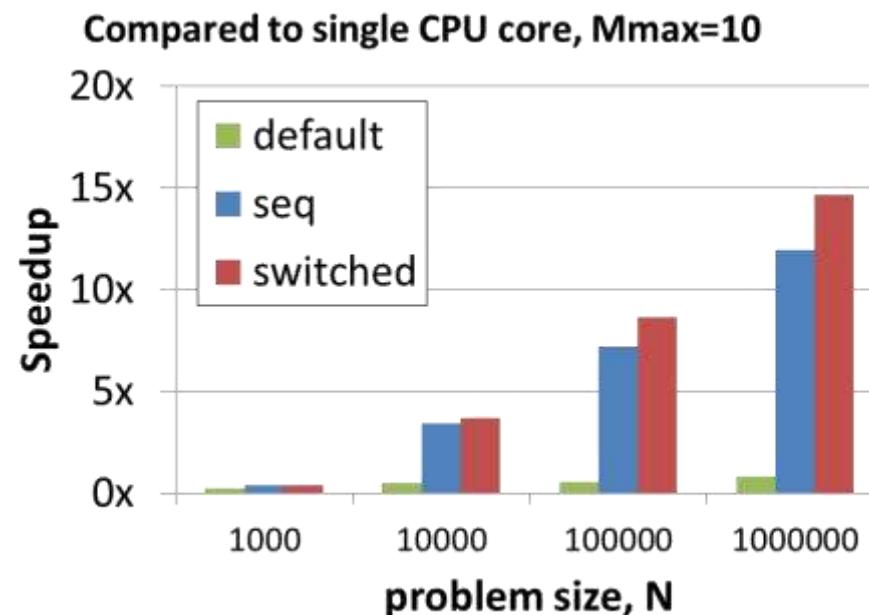
- Re-arrange the affected data arrays to allow coalescing
 - so warps get their data in the minimum number of vector loads/stores
- We want the **vector** index to be fastest-moving array index
 - switched array `ct(i,m)` now gives coalesced memory accesses
 - because `ct(1:32,m)` is a contiguous chunk of memory
- But this does mean refactoring your code
 - which may (or may not) be much work
 - Note: you may also get better CPU performance (e.g. with AVX instructions)

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

COMPUTE

| STO

Cray OpenAC



It's all vectorising, but still performing badly

- **Profile the code and start where it takes most time**
 - check the slowest thing really is compute kernels
 - if it really is a GPU compute kernels...
- **GPUs need lots of parallel tasks to work well**
 - check that there really are enough loop iterations
- **Then look at loop scheduling using OpenACC clauses**
- **Then might need to consider more extreme measures**
 - source code changes
 - handcoding CUDA kernels

Tuning the partitioning

- Before we alter the schedule, two easy tuning choices
 - number of threads per threadblock; total number of threadblocks
- Do this with clauses on the parallel/kernels directives
- **vector_length(<value>)**
 - changes the number of threads per threadblock
 - CCE-specific details:
 - value needs to be fixed at compile time
 - allowed values are: 1, 32, 64, 128, 256, 512, 1024
 - default value is 128
- **num_gangs(<value>)**
 - changes the number of threadblocks
 - you do not have to specify this (unlike with CUDA)
 - the compiler will make a default choice of sufficient blocks

Over-riding the default scheduling

- You can change how loops are scheduled
 - by adding additional **loop** directives above the relevant loops
 - with clauses to tell the compiler how to schedule that loop
 - use clauses: **gang**, **worker**, **vector**
 - we've already seen **vector** in use in the previous example
 - you can also use the **seq** clause
 - tells compiler what not to do, allowing it freedom in scheduling remaining loops
- Can partition a loop over multiple levels of parallelism
 - Put more than one of **gang**, **worker**, **vector** clauses on **loop** directive
 - To schedule one loop over whole device:
 - **acc loop gang worker vector**

Over-riding the default scheduling

- You can only mention **gang**, **worker**, **vector** (at most) once each per kernel
 - any not mentioned will be handled automatically by the compiler
 - it will choose where best to apply them
 - once all levels of parallelism are used up
 - additional loops will execute sequentially (redundantly)
- Handy tip:
 - To debug a kernel by running on a single GPU thread, use:
 - !\$acc parallel loop gang worker vector num_gangs(1) vector_length(1)
 - Useful for checking race conditions in parallelised loop nests
 - but execution will be very slow, so maybe find a cut-down testcase first

Aside: `vector_length` bigger than 32?

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

Advanced loop scheduling

- **OpenACC loop schedules are limited by the loop bounds**
 - one loop's iterations are divided over threadblocks
 - another loop's iterations are divided over threads in threadblock
- **This has limitations, for instance**
 - "tall, skinny" loopnests ($j=1:\text{big}$; $i=1:\text{small}$) won't schedule well
 - inner loop is too small
 - if less than 32 iterations won't even fill a warp, so wasted SIMT
 - "short, fat" loopnests ($j=1:\text{small}$; $i=1:\text{big}$) also not good
 - outer loop is too small
 - want lots of threadblocks to swap amongst SMs
 - In both cases there are enough **total** iterations to keep the GPU busy
 - but **division** of iterations between the two loops is non-optimal for the GPU
- **How do we better spread the loop iterations over the GPU?**
 - **collapse** clause
 - **worker** clause

collapse clause

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

Examples of using the collapse clause

- Consider a three-level loopnest (**i** inside **j** inside **k**)
 - loopnest should be perfectly nested to use **collapse** clause
- Here are a few examples of what you might do:
 - Collapse all three loops and schedule across GPU
 - "acc parallel loop collapse(3) gang worker vector" above **k**-loop
 - You don't really need "gang worker vector" here
 - there's only one composite loop left to partition after collapsing
 - Often the best way to improve performance of small tripcount loopnests
 - removes restriction of levels of parallelism tied to specific loops
 - Schedule inner two loops over threads in threadblock
 - "acc parallel loop gang" above **k**-loop
 - "acc loop collapse(2) vector" above **j**-loop
 - don't need "gang"; enough warps are used to cover all the iterations
 - Do this if the **k**-loop is quite large, but the **i**-loop is small
 - but you often see that **collapse(3)** is better

Explicit worker clauses

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

num_workers clause

- parallel/kernels can also take a num_workers clause
 - Clearly there is some overlap with vector_length clause
- Two examples of use:
- no worker clause but vector_length(NTHREADS>32)
 - is equivalent to specifying:
 - num_workers(NTHREADS/32)
 - plus vector_length(32)
 - so don't try and specify num_workers as well
- explicit worker clause
 - You probably only want to specify explicitly num_workers
 - only vector_length(32) makes sense in this case

collapse or worker?

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

The cache directive

- Main accelerator memory is relatively slow
 - If a data element is accessed multiple times
 - It may make sense to temporarily store it in a faster cache
 - typical use case is a finite difference stencil, e.g.
 - for all **i**, calculate: **a[i]=b[i+1]-2*b[i]+b[i-1]**
 - **b[3]** is used three times, to calculate all of: **a[2], a[3], a[4]**.
 - so we would like to cache values of **b** for reuse by other loop iterations
- The cache directive suggests this to the compiler
 - suggests, but does not compel: compiler can ignore suggestion
 - e.g. if you try to store more data than the cache can hold
 - check compiler feedback to see what it did
 - No guarantee that this improves performance
 - you may be polluting some automatic caching (hardware or software)
- Nvidia GPUs
 - threads within a threadblock execute on single SM piece of hardware
 - have joint access to a common, fast, but small, "shared memory"

COMPUTE | STORE | ANALYZE

cache clause example

- A first example:

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

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

Multi-dimensional loopnests

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

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

      A(i,j,k) = B(i, j, k) - &
                  ( B(i-1,j-1,k) &
                    + B(i-1,j+1,k) &
                    + B(i+1,j-1,k) &
                    + B(i+1,j+1,k) ) / 5

    ENDDO
  ENDDO
ENDDO
 !$acc end parallel
```

The **tile** clause

- Added to **loop** directive (new for OpenACC 2.0)
- Used to block loops in a loopnest
- Two main use cases:
 - To help the compiler implement the **cache** directive
 - Inner "element" loops have a fixed tripcount
 - So compiler knows how much shared memory is needed for **cache**
 - To allow compiler to use multidimensional threadblocks

The tile clause in more detail

Blocks loops in a loopnest

- Using specified blocking factors
 - fixed at compile-time
 - Or * to use compiler default
 - one argument per loop
- Outer tile loops
 - migrated to outside of nest
- Inner element loops
 - have known size

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

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

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

Scheduling clauses on **loop** directive still apply, if used

- gang** applies to outer, tile loops
- vector** applies to inner, element loops
 - probably want composite tilesize to be multiple of 32
- worker** might apply to either, depending on how used (see Standard)

Multi-dimensional caching

- The corrected version:

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

```
!$acc parallel loop gang
DO k = 1,NK
 !$acc loop tile(64,16) worker vector
 DO j = 1,NJ
     DO i = 1,NI
 !$acc cache( B(i-1:i+1,j-1:j+1,k) )
 
     A(i,j,k) = B(i, j, k) - &
                 ( B(i-1,j-1,k) &
                   + B(i-1,j+1,k) &
                   + B(i+1,j-1,k) &
                   + B(i+1,j+1,k) ) / 5
 ENDDO
 ENDDO
 ENDDO
 !$acc end parallel
```

Conclusions

- **Scheduling is the biggest optimisation**
 - after data movements
- **Does the code vectorise?**
 - Does it vectorise the right loop
- **Can collapsing loops help?**
- **Can workers help?**
- **Caching and tiling might be useful at the end**

OpenACC example: Tuning a simple code

Alistair Hart
Cray Exascale Research Initiative Europe



COMPUTE

|

STORE

|

ANALYZE

Tuning performance with the loop schedule

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

Tuning the code

- **Which kernels should we tune?**
 - Only one Himeno kernel is computationally intensive:
 - the finite-difference stencil loopnest
 - The other kernels were only ported to avoid data sloshing
 - We'll use code MFLOPS value as the performance metric
- **`vector_length` chooses number of threads per threadblock**
 - One of the easiest tuning choices
 - CCE: this needs to be fixed at compile time

Practical advice on tuning `vector_length`

- You probably have a lot of kernels to separately tune
 - Each `vector_length` choice requires a source edit and a recompile
 - Each kernel needs a separate performance metric
 - You want to minimise the amount of work
- My advice:
 - Use preprocessing to change the `vector_length` arguments:
 - insert clause: `vector_length(NTPB)`
 - set `NTPB` on compile: `ftn/cc -DNTPB=256 <source file>`
 - Put these clauses in right at the start of your OpenACC port
 - if there is any chance of you needing to tune later
 - Initially, use same argument (`NTPB`) for every kernel
 - Compile and run the code with profiling for different `NTPB`
 - remember to profile a production-sized problem
 - For each `NTPB` value, read off times for all the important kernels
 - For each important kernel, select the best `NTPB` value
 - Now alter source code to use best `NTPB` for each kernel

Scalar Himeno performance

- **PROBLEM_SIZE=1,2,3,4**

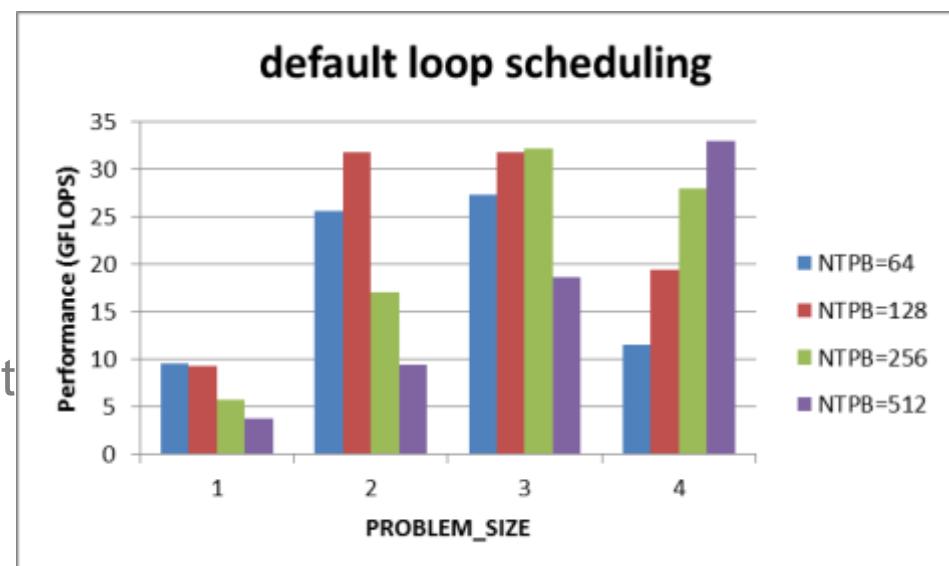
- For a parallel problem, this simulates strong scaling choices
 - increasing number of ranks decreases local (scalar) problem size
- Here we look at all 4 different sizes (all double precision)

- **vector_length choices:**

- NTPB = 64, 128, 256, 512

- **This has a BIG effect**

- best NTPB relative to default:
 - PROBLEM_SIZE=1: + 4%
 - PROBLEM_SIZE=2: default
 - PROBLEM_SIZE=3: + 1%
 - PROBLEM_SIZE=4: +69%



Changing loop scheduling

- This is the simplest algorithmic change

- Usual things to try
 - collapse or worker
- Which is best?
 - that's what we're trying to learn

- Which are good to try?
 - `collapse(2)` on inner two loops
 - this is then scheduled as vector
 - `collapse (3)` on all three loops
 - this is then scheduled over all GPU
 - `worker` for the middle loop

- Don't forget:
 - if you use extra `loop` directives
 - you must repeat `reduction` clauses

```
!$acc parallel loop red...
DO k = 2,kmax-1
 !$acc loop collapse(2) red...
 DO j = 2,jmax-1
 DO i = 2,imax-1
```

```
!$acc parallel loop red... &
 !$acc collapse(3)
 DO k = 2,kmax-1
 DO j = 2,jmax-1
 DO i = 2,imax-1
```

```
!$acc parallel loop red...
DO k = 2,kmax-1
 !$acc loop worker red...
 DO j = 2,jmax-1
 DO i = 2,imax-1
```

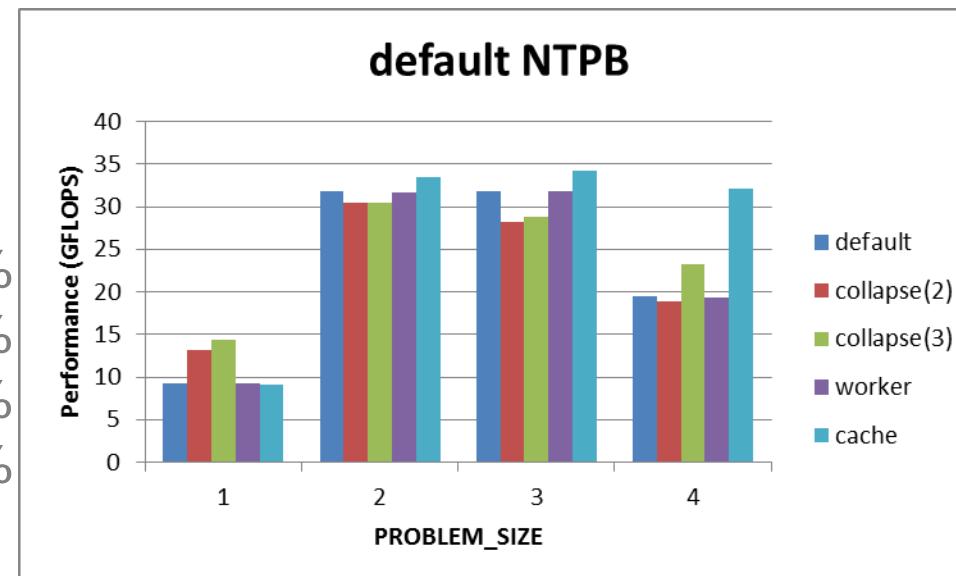
Using the cache clause

- Stencils share data between loop iterations
 - Good situation for trying **cache** clause
- Some choice in how to apply it
 - Maximum reuse:
 - `cache(p(i-1:i+1,j-1:j+1,k-1:k+1))`
 - but more shared memory used
 - and more time spent packing the cache
 - Progressively less:
 - `cache(p(i-1:i+1,j-1:j+1,k))`
 - `cache(p(i-1:i+1,j,k))`
- Which is best?
 - No definitive answer, but...
 - Just caching in i often seems best

```
!$acc parallel loop ...
DO k = 2,kmax-1
  DO j = 2,jmax-1
    DO i = 2,imax-1
      !$acc cache (p(i-1:i+1,j,k))
      S0=a(i,j,k,1)*p(i+1,j,k) &
```

Scalar Himeno performance

- PROBLEM_SIZE=1,2,3,4
- 5 algorithm choices for jacobi stencil kernel
 - default scheduling
 - collapse(2) inner i,j-loops
 - collapse(3) all loops
 - worker schedule for j-loop
 - cache(p(i-1:i+1,j,k)) with default loop scheduling
- vector_length: default
- Effect also quite big
 - best NTPB relative to default:
 - PROBLEM_SIZE=1: +55%
 - PROBLEM_SIZE=2: + 6%
 - PROBLEM_SIZE=3: + 8%
 - PROBLEM_SIZE=4: +65%

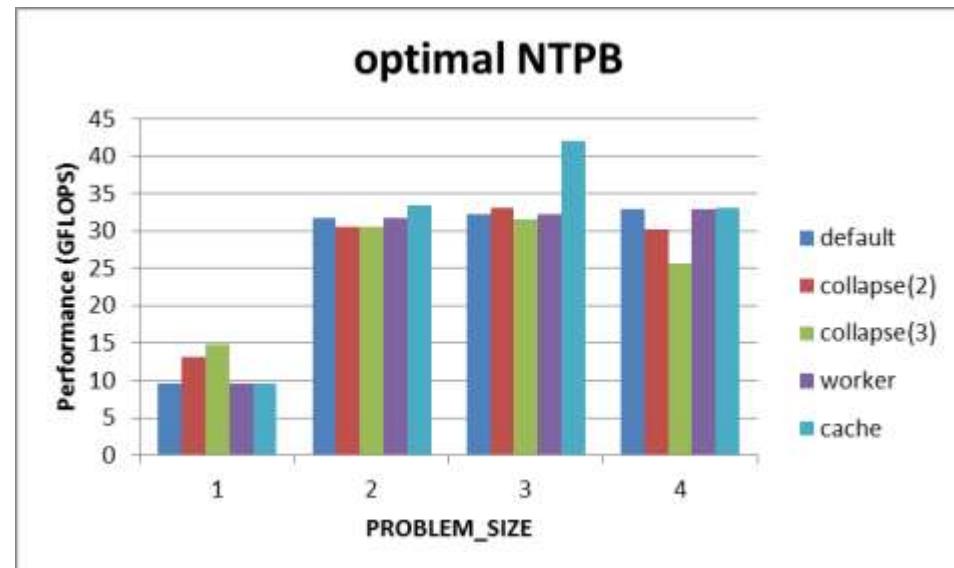


Just one tuning space

- **Really, we shouldn't tune these things separately**
 - For each algorithm choice, should find the optimal `vector_length`
 - (and repeat this for every problem size)
- **The parameter space gets very big, very quickly**
 - An autotuning framework may help here
- **Is it worth the effort?**
 - You might double the speed of a kernel
 - How would that affect your application's walltime?
 - Is this kernel's runtime the limiting factor in your application's performance?
- **But let's see what we can do with scalar Himeno...**

Scalar Himeno performance

- **PROBLEM_SIZE=1,2,3,4**
 - 5 algorithm choices for jacobi stencil kernel
 - choose best **vector_length** for each
- **Final improvements**
 - compared to default scheduling with default **vector_length**:
 - PROBLEM_SIZE=1: +60%
 - PROBLEM_SIZE=2: + 6%
 - PROBLEM_SIZE=3: +33%
 - PROBLEM_SIZE=4: +71%



Scalar Himeno tuning conclusions

- Tuning can have a big effect, relative to default
 - It is worth doing, but only after you optimise the data locality
 - data region gave **22x** speedup; kernel tuning gave less than **2x**
 - We did gain something at every problem size (don't reject a mere 2x!)
 - Only explored **vector_length** and basic scheduling (**collapse**, **worker**)
 - only change was OpenACC directives; CPU version same
- General conclusions for scalar Himeno:
 - Larger **vector_length** suited larger problem sizes
 - **64** best for PS1; **128** best for PS2; **256** best for PS3,4
 - Once we had optimised **vector_length**
 - **collapse(3)** was best for small problems (PS1)
 - **cache** was best for large problems (PS2-4)
 - caching in j,k directions did not improve performance

OpenACC 4: handling asynchronicity

Alistair Hart
Cray Exascale Research Initiative Europe



COMPUTE

| STORE

| ANALYZE

Exploiting overlap

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

Asynchronicity

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

Automatic synchronisation

- By default, the compiler will handle synchronisation
 - may be conservative:
 - and wait for every operation to complete
 - may be smarter,
 - and use a reduced number of waits consistent with correctness
- CCE-specific options:
 - control with compiler option `-hacc_model=auto_async_*`
 - `auto_async_none` :
 - waits for every operation to separately complete
 - useful for debugging and generating profiles
 - but it may skew performance
 - `auto_async_kernels` :
 - may allow some computational kernels to overlap
 - the default behaviour
 - `auto_async_all` :
 - may allow kernels and data transfers to overlap
 - try this as a performance-tuning option

User control via `async` clause

- **The `async` clause overrides the default behaviour**
 - User is now in control of synchronisation
 - applies to: `parallel`, `kernels`, `update` directives
 - User needs to insert appropriate synchronisation points
 - via `wait` directive or OpenACC API calls
 - **Beware:** there is no implicit `wait` at the end of a subprogram!
- **Between synchronisation points**
 - May get overlap of concurrently-executing kernels
 - depending on hardware
 - Should get overlap of computational kernels with data transfers
 - in both directions on PCIe bus
 - Data transfers in the same direction will not overlap
 - the runtime will automatically serialise the transfers
- **Queuing async operations in advance is a good idea**
 - even if the operations themselves are serialising, get better throughput

A stream of tasks

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

Multiple streams of tasks

- You can launch multiple streams of tasks at once
 - each with a different handle value
 - Tasks within a given stream
 - guaranteed to execute sequentially
 - Tasks in different streams
 - may overlap or serialise, as the hardware and runtime allows
 - operations in different streams should be independent
 - or we have a race condition
- Synchronisation
 - `wait(handle)` directive ensures one stream has completed
 - `wait(handle1,handle2,...)` can be used to finalise multiple streams
 - `wait` directive with no argument ensures all streams have completed
 - API calls can be used to wait on, or test for, all these completion cases
- Nvidia GPUs (Kepler, Fermi)
 - currently support up to 16 simultaneous streams in hardware
 - if handle is too large, runtime MODs it back into allowable range
 - so handle=16 same as handle=0
 - this can lead to false dependencies between streams

OpenACC async first example

- **a simple pipeline:**

- processes an array, slice by slice
- each slice requires 3 tasks:
 - copy data to GPU,
 - process on GPU,
 - bring back to CPU
- which must execute sequentially
- but we can overlap different slices
- Use a different stream for each slice
 - use slice number as stream handle
 - don't worry if number gets too large
 - OpenACC runtime maps it back into allowable range (using MOD function)

```

REAL :: 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

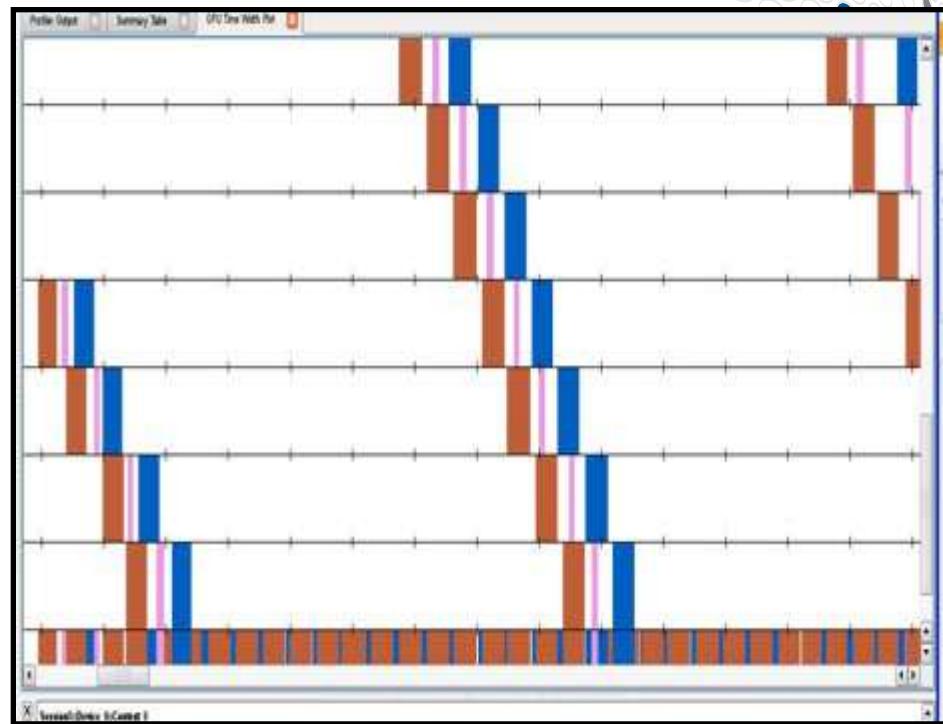
```

- **Expect to see overlap of three streams at once**

- one sending to the device; one processing the slice; one sending to host

OpenACC async results

- Execution times:
 - CPU: 3.76s
 - OpenACC: 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 device
 - pink: computational on device
 - blue: data transfer to host
 - 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)



COMPUTE

STORE

ANALYZE

Going further with OpenACC 2.0

● OpenACC 1.0:

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

● OpenACC 2.0:

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

High-level example

```

!$acc parallel loop async(stream1)
<Kernel A>
!$acc parallel loop async(stream1)
<Kernel B>

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

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

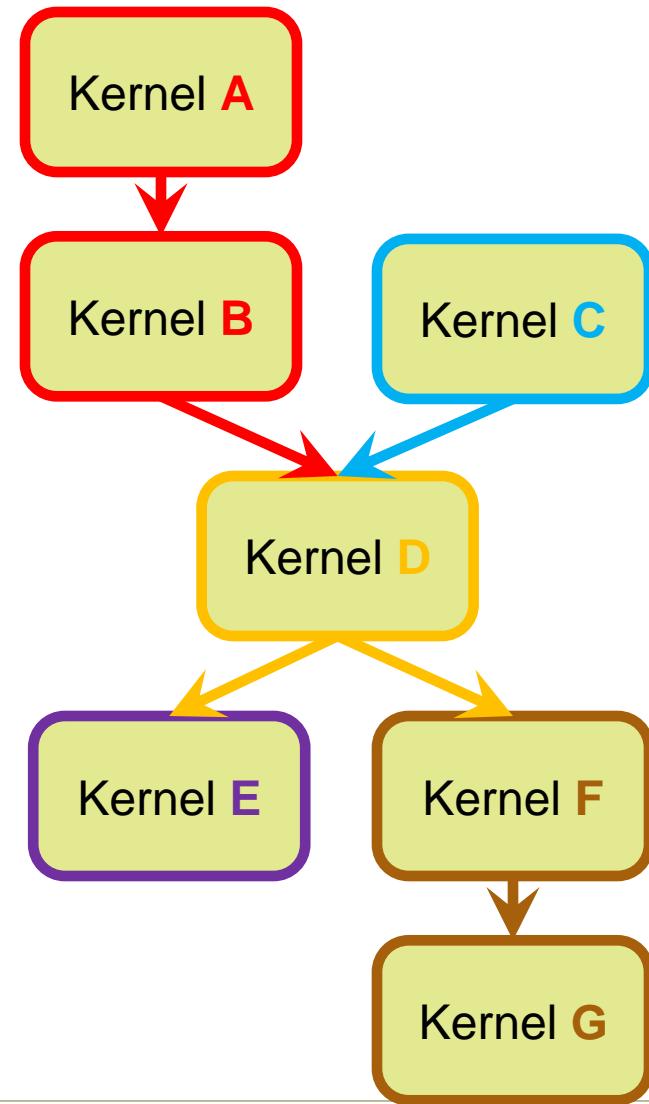
!$acc parallel loop async(stream4) &
!$acc      wait(stream3)
<Kernel E>

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

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

!$acc wait ! ensures all completed

```



STORE

ANALYZE

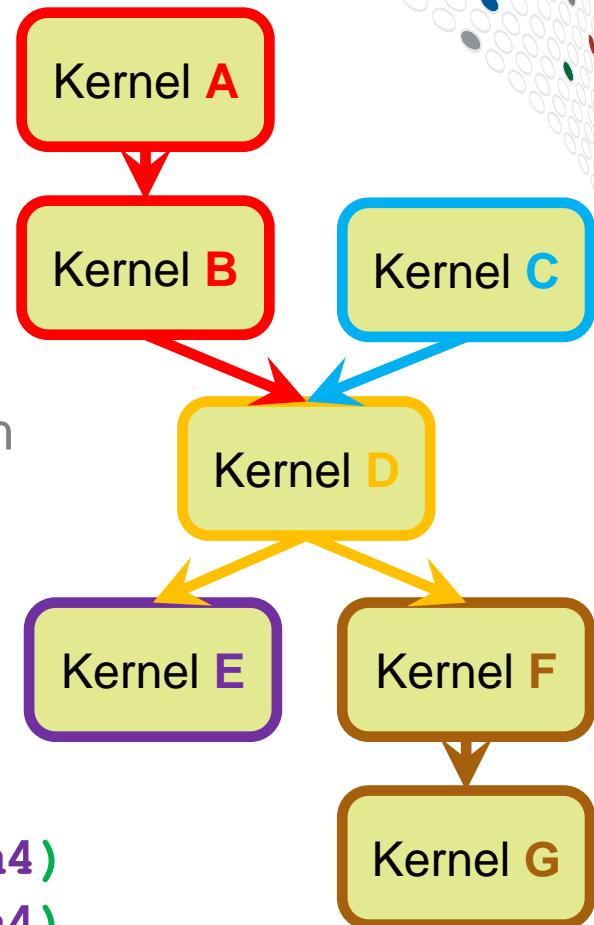
Advanced asynchronous concepts

- Could reuse streams

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

- Asynchronous waits ?!

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



Using dependency trees

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

OpenACC case study: Porting a parallel code

Alistair Hart
Cray Exascale Research Initiative Europe



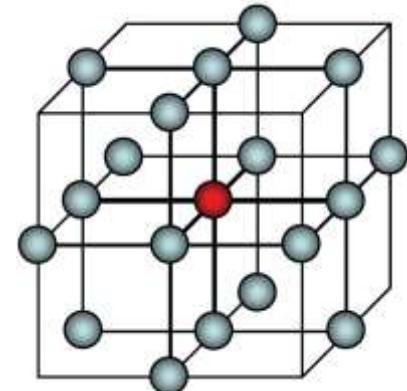
COMPUTE | STORE | ANALYZE

Overview

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

Contents

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



The Himeno Benchmark

- **Parallel 3D Poisson equation solver**
 - Iterative loop evaluating 19-point stencil
 - Memory intensive, memory bandwidth bound
- **Fortran, C, MPI and OpenMP implementations available from <http://accr.riken.jp/2444.htm>**
 - Cray also has a Fortran coarray (CAF) version
- **Productivity of OpenACC**
 - ~600 lines of Fortran
 - Fully ported to accelerator using around 30 directive pairs

Overall program structure

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

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

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

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

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

  xmlops2 = flop*1.0d-6/cpu*nn

  PRINT *, ' Loop executed ',nn,' times'
  PRINT *, ' Gosa : ',gosa
  PRINT *, ' MFLOPS: ',xmlops2,' time(s): ',cpu
END PROGRAM himeno
```

The distributed jacobi routine

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

```
iter_lp: DO loop = 1,nn  
  
    compute stencil: wrk2, wgosa from p  
  
    pack halos from wrk2 into send bufs  
  
    exchange halos with neighbour PEs  
  
    sum wgosa across PEs  
  
    copy back wrk2 into p  
  
    unpack halos into p from recv bufs  
  
ENDDO iter_lp
```

The Jacobi computational kernel (serial)

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

```

DO K=2,kmax-1
DO J=2,jmax-1
DO I=2,imax-1
  s0=a(I,J,K,1)* p(I+1,J, K )
    +a(I,J,K,2)* p(I, J+1,K ) &
    +a(I,J,K,3)* p(I, J, K+1) &
    +b(I,J,K,1)*(p(I+1,J+1,K )-p(I+1,J-1,K ) &
      -p(I-1,J+1,K )+p(I-1,J-1,K )) &
    +b(I,J,K,2)*(p(I, J+1,K+1)-p(I, J-1,K+1) &
      -p(I, J+1,K-1)+p(I, J-1,K-1)) &
    +b(I,J,K,3)*(p(I+1,J, K+1)-p(I-1,J, K+1) &
      -p(I+1,J, K-1)+p(I-1,J, K-1)) &
    +c(I,J,K,1)* p(I-1,J, K ) &
    +c(I,J,K,2)* p(I, J-1,K ) &
    +c(I,J,K,3)* p(I, J, K-1) &
    + wrk1(I,J,K)

  s = (s0*a(I,J,K,4)-p(I,J,K))*bnd(I,J,K)
  wgosa = wgosa + ss*ss
  wrk2(I,J,K) = p(I,J,K) + omega * ss
ENDDO
ENDDO
ENDDO

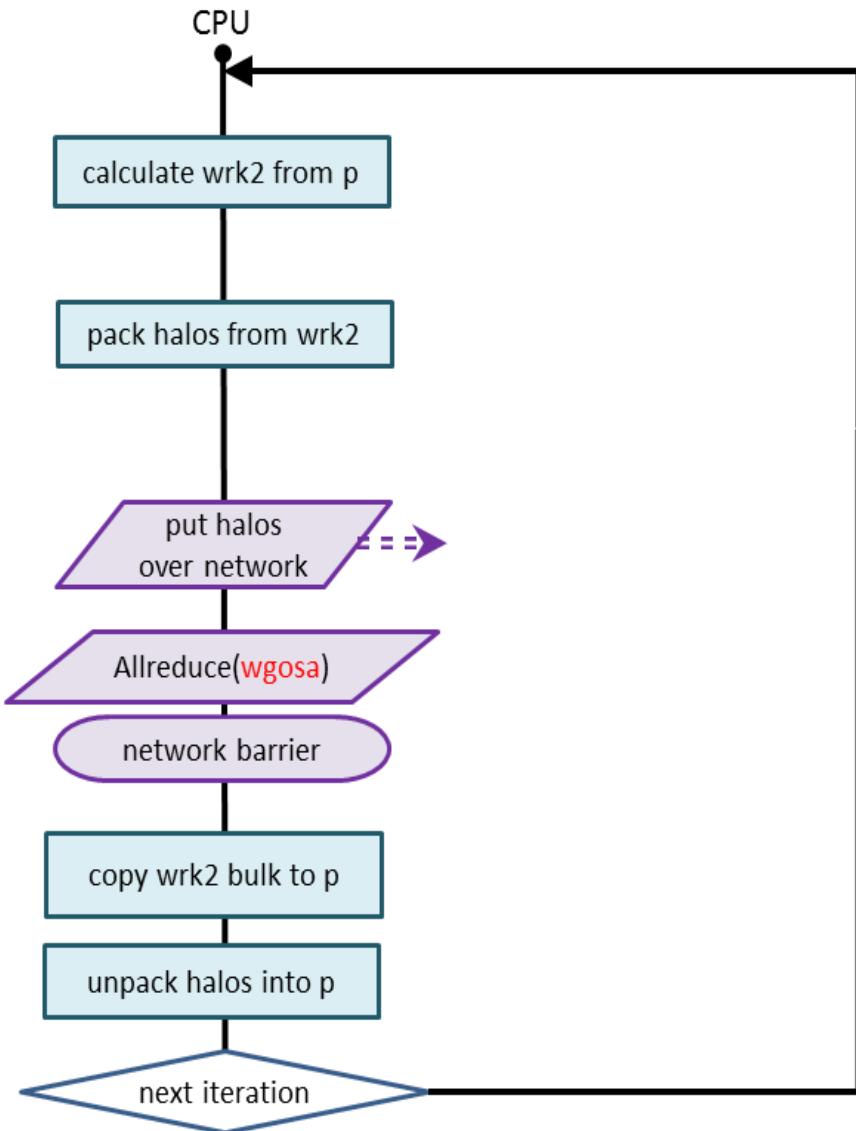
```

COMPUTE

S

Distributed jacobi routine as a flowchart

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



First OpenACC port

- **Loopnest kernels**

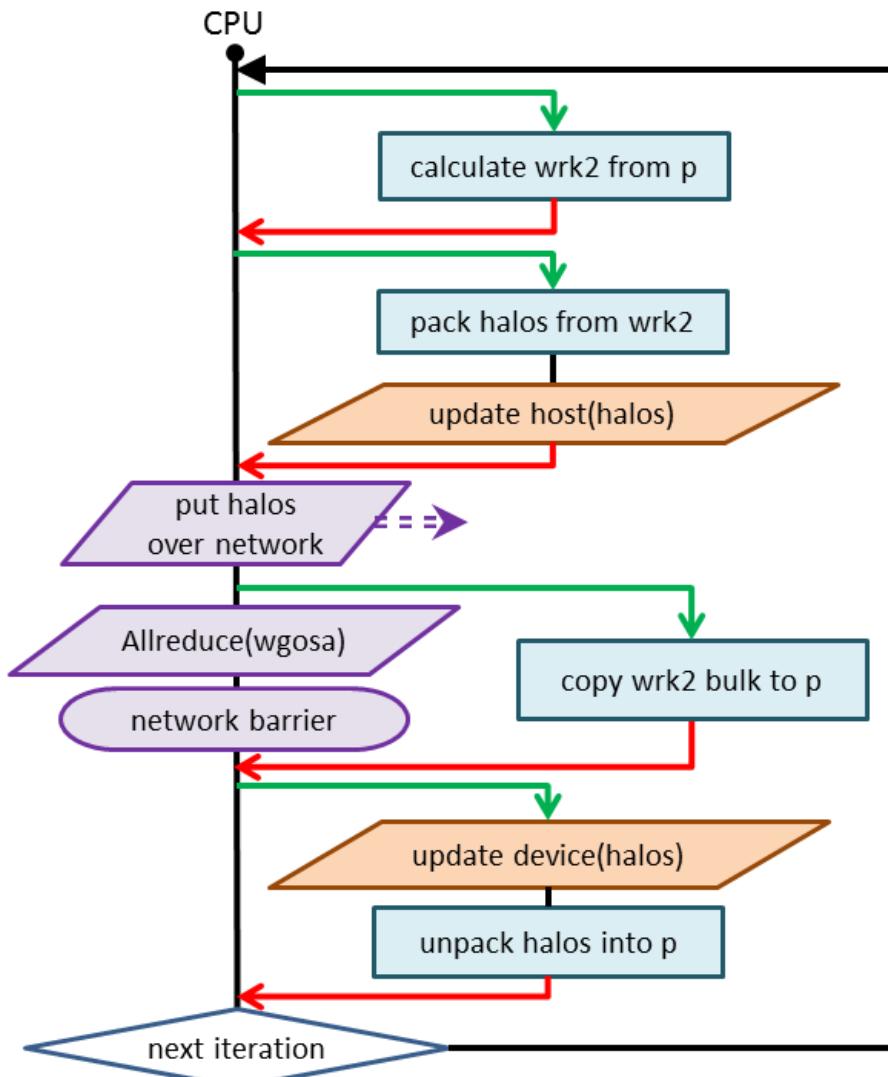
- stencil
- pack halos: x , y , z
- wrk2 \rightarrow bulk
- unpack halos: x , y , z

- **acc updates**

- 6 buffers each time

- **Some additional overlap**

- GPU kernels asynchronous
- wrk2 \rightarrow bulk
 - can overlap with comms



COMPUTE

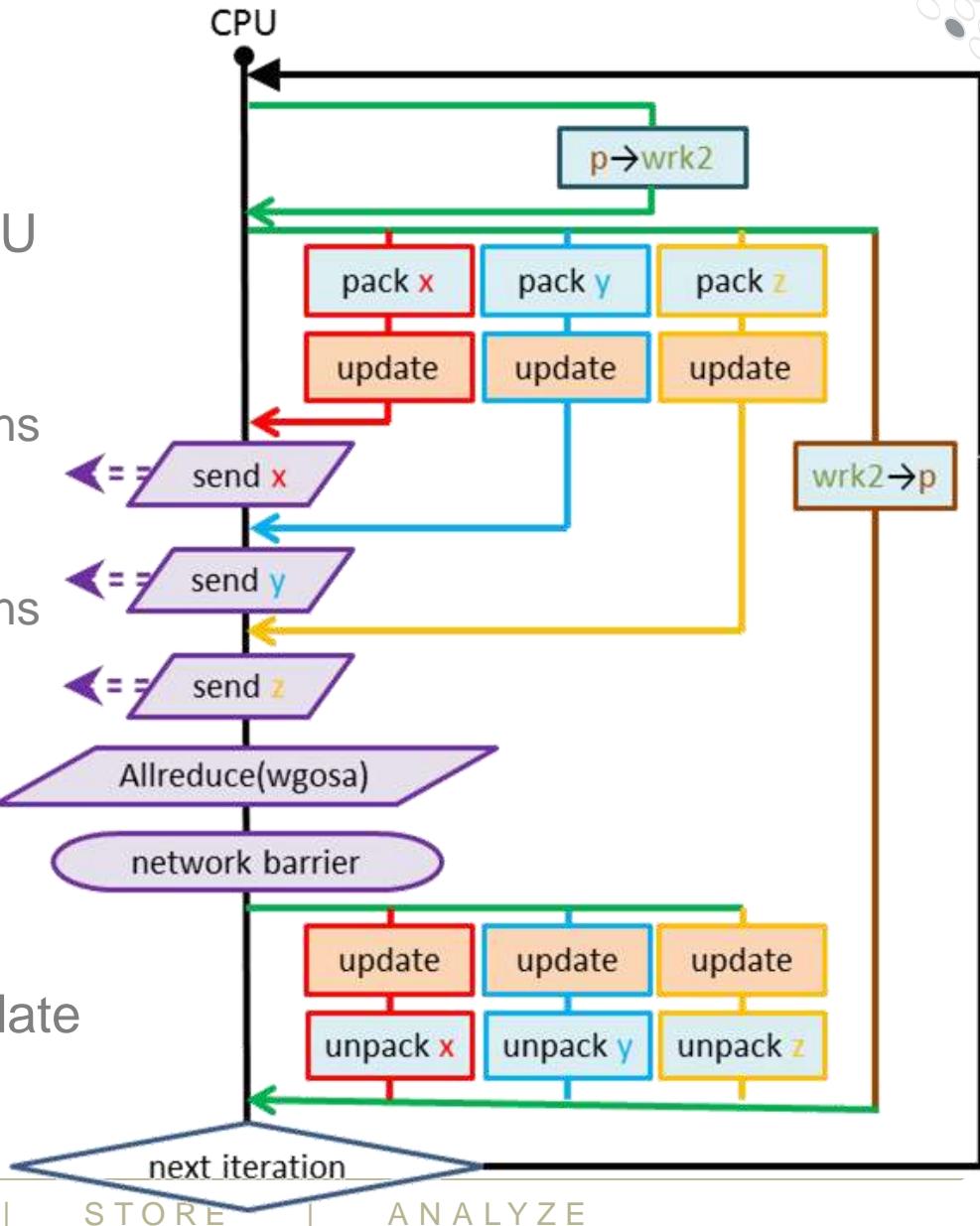
Asynchronicity

- **async clause**

- allows task overlap on GPU
- halo pack/update
 - can overlap **x**, **y**, **z** directions
- halo update/unpack
 - can overlap **x**, **y**, **z** directions
- can also overlap **wrk2**→**p**

- **Host/GPU sync points**

- After first kernel
- After each buffer pack/update
- At the end of the iteration



Packing and transferring send buffers

• Use **async** clause

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

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

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

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

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

Packing and transferring send buffers (better)

- The biggest bottleneck

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

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

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

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

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

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

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

Transferring and unpacking recv buffers

• Copying wrk2 into p

- very quick kernel
- but can overlap with network
 - could start this kernel before we sent the data

• When messages complete

- copy and unpack recv buffers
- three parallel streams again

• wait

- ensures all 4 streams complete
 - 3 update/unpacks plus bulk
- ready to start next iteration

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

<network barrier>

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

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

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

 !$acc wait

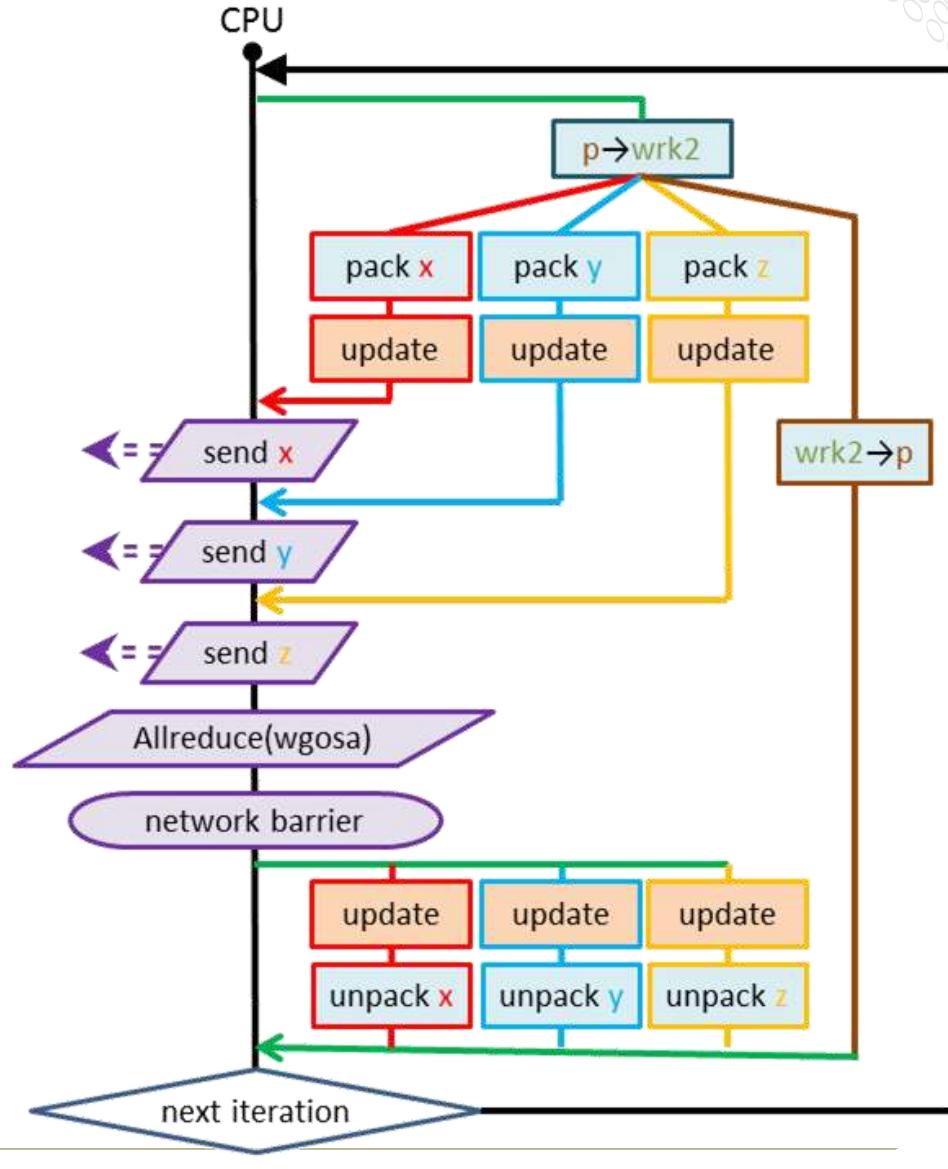
<end iteration loop>
```

Going further with OpenACC v2.0

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

Using the OpenACC v2.0 `wait` clause

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



COMPUTE

STORE

ANALYZE

Using the OpenACC v2.0 wait clause

- In more detail:

- Introduce new stream

- jstrm for stencil kernel
- buffer-packing streams wait on this completing
- acc wait(xstrm) guarantees jstrm has also completed
 - (also waits on ystrm, zstrm)

- bulk copy kernel also waits on jstrm

- we don't set jstrm=bstrm
- want to kick-off buffer packs as soon as wrk2 is ready

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

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

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

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

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

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

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

Data region considerations

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

Communications models

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

MPI

● Use nonblocking MPI calls

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

Better host-side MPI

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

Improved G2G MPI

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

Using the G2G MPI

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

```

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

<stencil p -> wrk2> async(jstrm)

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

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

call MPI_WAITALL(12,...)

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

!$acc wait

```

Summary

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

Summary

- **Some things to look for**

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

- **The same principles apply more widely**

- to other message-passing programming models
- CAF, SHMEM etc.

OpenACC 5: advanced topics

Alistair Hart
Cray Exascale Research Initiative Europe

Contents

- Some more advanced OpenACC topics

- Advanced data management
 - `enter/exit data` constructs
 - `declare` directive
 - Some “examples”
- Separate compilation support
 - `routine` construct
- Device specialisation
 - `device_type` clause
- nested parallelism
- runtime routines
- Interoperability feature
 - `host_data` construct
 - CUDA interoperability example

Unstructured data lifetimes

Enter Data construct

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

Exit Data construct

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

Global Object management

- **!\$acc declare**

- Makes a variable resident in accelerator memory
 - persists for the duration of the implicit data region
- Makes a variable available in accelerator code for linking

- **clauses**

- **device_resident(obj-list)**
 - Place object on device as a global, only device copy is required
- **link(obj-list)**
 - Place a pointer in device global memory for the listed objects
 - Programmer is responsible for actually moving object to the device

Data management features and global data

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

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

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

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

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

Data management features

unstructured data lifetimes

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

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

...

```
#pragma acc exit data delete(a[0:n])
```

...

```
#pragma acc exit data copyout(b[0:n])
```

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

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

Procedure calls, separate compilation

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

Routine directive examples

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

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

Device-specific tuning, multiple devices

- device_type(dev-type)

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

Nested Parallelism

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

The OpenACC Runtime API

The OpenACC runtime API

- **Directives are comments in the code**
 - automatically ignored by non-accelerating compiler
- **OpenACC also offers a runtime API**
 - set of library calls, names starting `acc_`
 - set, get and control accelerator properties
 - offer finer-grained control of asynchronicity
 - Data allocation and movement
 - OpenACC specific
 - will need pre-processing away for CPU execution
 - `#ifdef _OPENACC`
- **Advice: you do not need the API for most codes.**
 - Start without it, only introduce it where it is really needed.
 - I almost never use it

Runtime API for Device Selection and Control

● About the OpenACC-supporting accelerators

- Note: none of these are relevant for a Cray XC30 or XK7
- What type of device will I use next? `acc_get_device_type()`
 - default from environment variable `ACC_DEVICE_TYPE`
- What type of device should I use next? `acc_set_device_type()`
- How many accelerators of specified type? `acc_get_num_devices()`
- Which device of named type will I use next? `acc_get_device_num()`
 - default from environment variable `ACC_DEVICE_NUM`
- Set device of named type to use next? `acc_set_device_num()`
- Am I executing on device of specified type? `acc_on_device()`

● Initialising/shutting down accelerators:

- Initialise (e.g. to isolate initialisation time): `acc_init()`
- Shut down (e.g. before switching devices): `acc_shutdown()`

Runtime API for Advanced Memory Control

- Offer method to allocate and free device memory
 - These are for very advanced users
 - Replicate the functionality of data regions
- C/C++ only:
 - `void* acc_malloc (size_t);`
 - `void acc_free (void*);`
- All languages (C/C++ shown)
 - `acc_copyout(void*, size_t);`
 - `acc_delete(void*, size_t);`
 - `acc_is_present(void*, size_t);`
 - `acc_[present_or_]copyin(void*, size_t);`
 - `acc_[present_or_]create(void*, size_t);`
 - `acc_update_device(void*, size_t);`
 - `acc_update_self(void*, size_t);`
- If you just need to know the location of OpenACC memory
 - for instance, to pass this address to CUDA
- Then you don't need these API calls
 - just use `host_data` directive or `acc_deviceptr` instead

Advanced Memory management

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

Runtime API for Asynchronicity

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

host_data directive

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

COMPUTE

STORE

ANALYZE

Interoperability with CUDA

```

PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$acc data copy(a)
! <Populate a(:) on device
!   as before>
!$acc host_data use_device(a)
  CALL dbl_cuda(a)
!$acc end host_data
!$acc end data
  <stuff>
END PROGRAM main

```

```

__global__ void dbl_knl(int *c) {
  int i = \
    blockIdx.x*blockDim.x+threadIdx.x;
  if (i < N) c[i] *= 2;
}

extern "C" void dbl_cuda_(int *b_d) {
  cudaThreadSynchronize();
  dbl_knl<<<NBLOCKS,BSIZE>>>(b_d);
  cudaThreadSynchronize();
}

```

- **host_data** region exposes accelerator memory address on host
 - nested inside **data** region
- Call **CUDA-C wrapper (compiled with nvcc; linked with CCE)**
 - must include **cudaThreadSynchronize()**
 - Before: so asynchronous accelerator kernels definitely finished
 - After: so **CUDA** kernel definitely finished before we return to the **OpenACC**
 - **CUDA** kernel written as usual
 - Or use same mechanism to call existing **CUDA** library

Summary

- **A lot of the features in this talk are quite advanced**
 - They give the user a great deal of control
 - And allow more sophisticated codes to be ported with OpenACC
- **But for many codes, you don't need these at the start**
 - We ported our examples without using these features
 - and got good performance
 - You might want to consider them later
 - if you can port a first version using the basic directives

Porting a larger code

Alistair Hart
Cray Exascale Research Initiative Europe



COMPUTE

STORE

ANALYZE

Adding OpenACC to a Larger Code

- **Adding OpenACC to a real code is not trivial work...**
 - Are parts of the program suitable for an accelerator?
 - Where do we start?
 - What do we do next?
- **We'll go through the exercise for an example code**
 - Running on a Cray XC30 (AMD Interlagos and Nvidia Kepler K20x)
 - Using Cray compiler and Cray performance analysis tools

The Code

• NAS Parallel Benchmarks MG (MultiGrid) code

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

Building and Running MG

● Build:

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

● Run:

- Three important lines of output

- Fortran

- L2 Norm is 0.1800564401355E-05

- Mop/s total = 4787.41 ! Fortran

- Verification = SUCCESSFUL

- C output same, but baseline performance differs

- Mop/s total = 4848.87 // C

- Always check:

- L2 Norm should not be a NaN

- Verification should be successful

Where Do We Start?

- Profile MG on the CPU

Table 1: Profile by Function Group and Function

Time%	Time	Imb. Time	Imb. Time%	Calls	Group Function
100.0%	4.213820	--	--	1630.0	Total
<hr/>					
100.0%	4.213768	--	--	1230.0	USER
<hr/>					
47.8%	2.013292	--	--	161.0	resid_
23.9%	1.008719	--	--	160.0	psinv_
14.2%	0.599713	--	--	140.0	rprj3_
10.7%	0.450218	--	--	140.0	interp_
2.1%	0.089875	--	--	461.0	comm3_
<hr/>					

- Four routines dominate the runtime

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

COMPUTE | STORE | ANALYZE

Understand Flow of the Application

Table 1: Function Calltree View

Time%	Time	Calls	Calltree
100.0%	4.213820	1630.0	Total
100.0%	4.213768	1230.0	mg_
76.0%	3.202582	1180.0	mg3p_
3	25.5%	1.074038	280.0 resid_
3	24.6%	1.038074	320.0 psinv_
3	14.4%	0.605792	280.0 rprj3_
3	10.7%	0.450218	140.0 interp_
23.6%	0.993695	42.0	resid_

● mg calls:

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

COMPUTE | STORE | ANALYZE

Get Work Estimates for Loops

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

Loop Incl Time Total	Loop Hit	Loop Trips Avg	Loop Trips Min	Loop Trips Max	Function=/.LOOP[.]
<hr/>					
2.073902	161	96.497	4	256	resid_.LOOP.1.li.634
2.073579	15536	201.067	4	256	resid_.LOOP.2.li.635
1.013344	3123776	235.548	4	256	resid_.LOOP.4.li.642
0.972528	3123776	237.548	6	258	resid_.LOOP.3.li.636

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

Add First OpenACC Kernel

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

Resulting MG Performance?

- Running with and without OpenACC kernel:

	Original (Mop/s)	1 kernel (Mop/s)
Fortran	4787	3509
C	4849	3597

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

Enable Cray Runtime Commentary

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

```
ACC: Start transfer 6 items from mg_v02.f:615
ACC:     allocate, copy to acc 'a' (32 bytes)
ACC:     allocate 'r' (137388096 bytes)
ACC:     allocate, copy to acc 'u' (137388096 bytes)
ACC:     allocate, copy to acc 'v' (137388096 bytes)
ACC:     allocate <internal> (530432 bytes)
ACC:     allocate <internal> (530432 bytes)
ACC: End transfer (to acc 274776224 bytes, to host 0 bytes)
ACC: Execute kernel resid_ckpt_L615_1 blocks:256 threads:128 async(auto) from mg_v03.f:615
ACC: Wait async(auto) from mg_v02.f:639
ACC: Start transfer 6 items from mg_v02.f:639
ACC:     free 'a' (32 bytes)
ACC:     copy to host, free 'r' (137388096 bytes)
ACC:     free 'u' (137388096 bytes)
ACC:     free 'v' (137388096 bytes)
ACC:     free <internal> (0 bytes)
ACC:     free <internal> (0 bytes)
ACC: End transfer (to acc 0 bytes, to host 137388096 bytes)
```

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

COMPUTE | STORE | ANALYZE

Or Use Nvidia Compute Profiler

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

```
method=[ memcpyHtoD ] gputime=[ 1.248 ] cputime=[ 6.727 ]
method=[ memcpyHtoD ] gputime=[ 23297.504 ] cputime=[ 23345.885 ]
method=[ memcpyHtoD ] gputime=[ 23329.568 ] cputime=[ 23329.770 ]
method=[ resid_ck_L615_1 ] gputime=[ 7611.488 ] cputime=[ 16.093 ] occupancy=[ 0.312 ]
method=[ memcpyDtoH ] gputime=[ 63877.535 ] cputime=[ 64378.523 ]
```

- Data transfers obvious, taking most time

Or Use CrayPAT for a Profile by Function

Table 1: Profile by Function Group and Function

Time%	Time	Imb.	Imb.	Calls	Group
		Time	Time%		Function
100.0%	6.981196	--	--	1252.0	Total
100.0%	6.981093	--	--	851.0	USER
48.2%	3.364813	--	--	1.0	mg_
31.2%	2.175330	--	--	170.0	resid_.ACC_COPY@li.615
15.3%	1.068125	--	--	170.0	resid_.ACC_COPY@li.639
5.3%	0.368549	--	--	170.0	resid_.ACC_SYNC_WAIT@li.639
0.1%	0.003585	--	--	170.0	resid_.ACC_ASYNC_KERNEL@li.615
0.0%	0.000691	--	--	170.0	resid_.ACC_REGION@li.615
0.0%	0.000103	--	--	401.0	ETC

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

COMPUTE | STORE | ANALYZE

...And CrayPAT Accelerator Statistics

Table 2: Time and Bytes Transferred for Accelerator Regions

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

- Host and accelerator times given separately

- ACC_KERNEL
 - Acc Time is the compute time
 - Host Time is the time for the asynchronous launch
 - The Host "catches up" at the SYNC_WAIT

... And CrayPAT Summarized Trace

- `pat_build -u mg.B.x`

Table 1: Profile by Function Group and Function

Time%	Time	Imb.	Imb.	Calls	Group	Function
		Time	Time%			
100.0%	7.005544	--	--	265301.0	Total	
100.0%	7.005442	--	--	264900.0	USER	
31.1%	2.176236	--	--	170.0	resid_.ACC_COPY@li.615	
15.2%	1.067545	--	--	170.0	resid_.ACC_COPY@li.639	
15.2%	1.066457	--	--	168.0	psinv_	
11.6%	0.811638	--	--	131072.0	vranlc_	
9.1%	0.637047	--	--	147.0	rprj3_	
6.7%	0.469986	--	--	147.0	interp_	
5.3%	0.368538	--	--	170.0	resid_.ACC_SYNC_WAIT@li.639	
1.5%	0.103028	--	--	149.0	zero3_	
1.5%	0.102412	--	--	2.0	zran3_	
1.4%	0.098443	--	--	487.0	comm3_	

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

COMPUTE | STORE | ANALYZE

More OpenACC Kernels

- Running with 4 accelerated kernels:

	Original (Mop/s)	1 kernel (Mop/s)	4 kernels (Mop/s)
Fortran	4787	3509	2727
C	4849	3597	1727

- Even slower, and C particularly bad. Why?

Profile the Code Again

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

```
704.         #pragma acc parallel loop private(z1,z2,z3) \
705.             copy(u[0:n1*n2*n3])
706.             copyin(z[0:mm1*mm2*mm3])
707. + gG-----<   for (i3 = 0; i3 < mm3-1; i3++) {
708. + gG 2-----<       for (i2 = 0; i2 < mm2-1; i2++) {
709. gG 2 g----<           for (i1 = 0; i1 < mm1; i1++) {
710. gG 2 g
711. gG 2 g           i123 = i1 + mm1*i2 + mm12*i3;
712. gG 2 g           z1[i1] = z[i123+mm1] + z[i123];
713. gG 2 g           z2[i1] = z[i123+mm12] + z[i123];
714. gG 2 g---->       z3[i1] = z[i123+mm1+mm12] + z[i123+mm12] + z1[i1];
715. + gG 2 r4---<       }
716. gG 2 r4           for (i1 = 0; i1 < mm1-1; i1++) {
717. gG 2 r4           i123 = i1 + mm1*i2 + mm12*i3;
718. gG 2 r4           j123 = 2*i1 + n1*(2*i2 + n2 * 2*i3);
719. gG 2 r4           u[j123] += z[i123];
720. gG 2 r4           u[j123+1] += 0.5*(z[i123+1]+z[i123]);
```

Loop Accelerated

Loop Partitioned

Loop Not Partitioned

- Loop at line 715 not partitioned
 - Executed redundantly: every thread does every loop iteration

More OpenACC Kernels

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

	Original (Mop/s)	1 kernel (Mop/s)	4 kernels (Mop/s)
Fortran	4787	3509	2727
C	4849	3597	2715

- Fortran and C performance now identical

Next Steps – Reduce Data Movement

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

Results for Accelerating up the Calltree

- Accelerated loops in norm2u3, zero3, comm3

	Original (Mop/s)	1 kernel (Mop/s)	4 kernels (Mop/s)	Calltree Routines (Mop/s)
Fortran	4787	3509	2727	1884
C	4849	3597	2715	1821

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

	Original (Mop/s)	1 kernel (Mop/s)	4 kernels (Mop/s)	Calltree Routines (Mop/s)
Fortran	4787	3509	2727	1884
C	4849	3597	2715	1878

COMPUTE | STORE | ANALYZE

Add Data Region

- Now we put a data region in the main program

- Arrays u,v,r are declared **create**
 - We'll never use the host version of these
- Arrays a,c are declared **copyin**
 - They're initialized on the host

- Then in all the subprograms, we change clauses

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

Results with Data Region in Main

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

	Original (Mop/s)	1 kernel (Mop/s)	4 kernels (Mop/s)	Calltree Routines (Mop/s)	Data Region (Mop/s)
Fortran	4787	3509	2727	1884	27583
C	4849	3597	2715	1878	26476

View Compiler Commentary Again

- All array data transfers eliminated

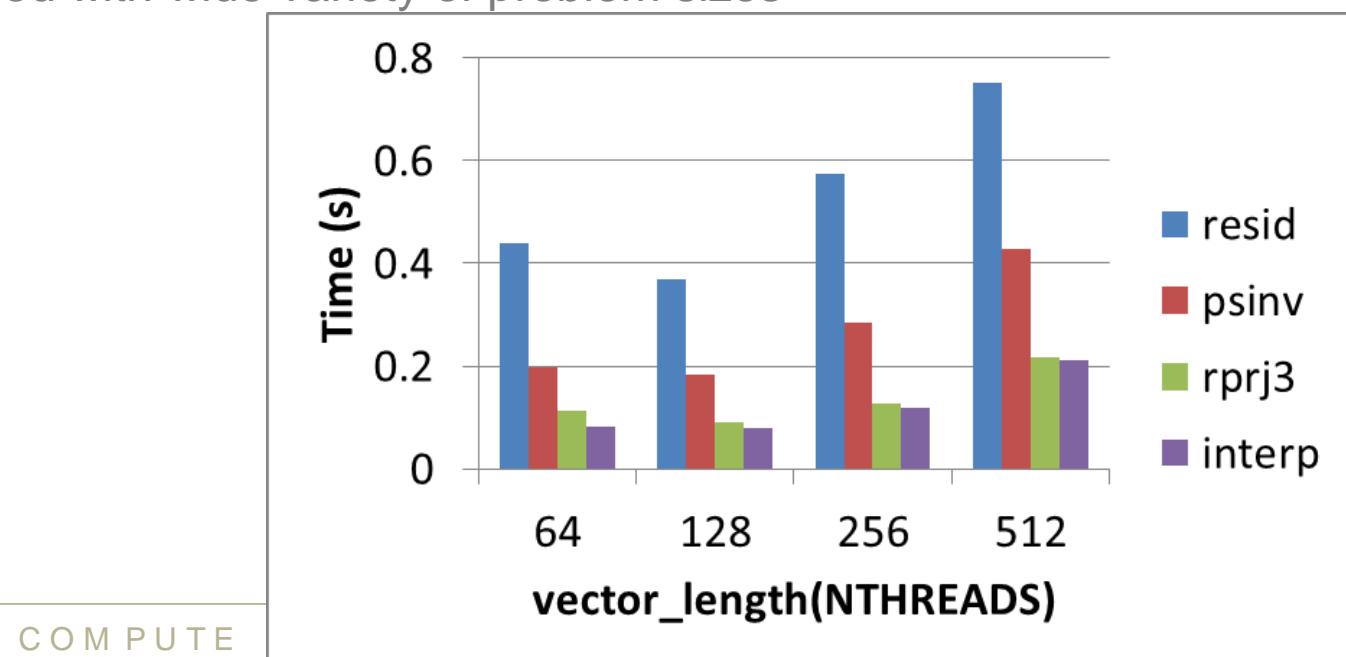
- Run with CRAY_ACC_DEBUG=2 and catch STDERR in a file
- grep "copy" <file> | sort | uniq

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

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

Performance Tuning Tips

- Check the .lst loopmark file
 - Are any kernels obviously badly scheduled?
 - No, we already checked that
- Try varying `vector_length` from the default of 128
 - Different values may suit different kernels
 - No effect here: 128 is the best choice
 - Multigrid complicates things;
 - routines called with wide variety of problem sizes



Performance Tuning for resid()

- Five main tuning options

- Loop restructuring
 - Single, perfectly-nested loopnest
 - removing u1 array by manually inlining
 - temporary arrays could be in global memory
 - Two full loopnests
 - explicitly privatised temporaries: $u1(i,j,k)$
 - perfectly-nested loopnests tend to schedule better

- Loop scheduling
 - collapse** perfectly-nested loops
 - increases flexibility in scheduling loop iterations
 - worker** clause on j-loop
 - also increases the amount of parallelism mapped to the accelerator threads
- Caching re-used data
 - How much: $u(i-1:i+1,j,k)$ or $u(i-1:i+1,j-1:j+1,k)$ or $u(i-1:i+1,j-1:j+1,k-1:k+1)$
- Tiling the loopnest
 - especially if we are using the **cache** clause
- Varying **vector_length** (number of threads per block)

```

DO k = 2,n3-1
DO j = 2,n2-1
DO i = 1,n1
  u1(i) = <j,k-stencil on u>
ENDDO
DO i = 2,n1-1
  r(i,j,k) = <i-stencil on u1>
ENDDO
ENDDO
ENDDO

```

So which is best?

- Lots of options to explore

- best option likely to depend on problem size
 - which varies in a single MG run as we cycle through the blockings
- may need different algorithm versions for different problem sizes

- Fortunately, in this case:

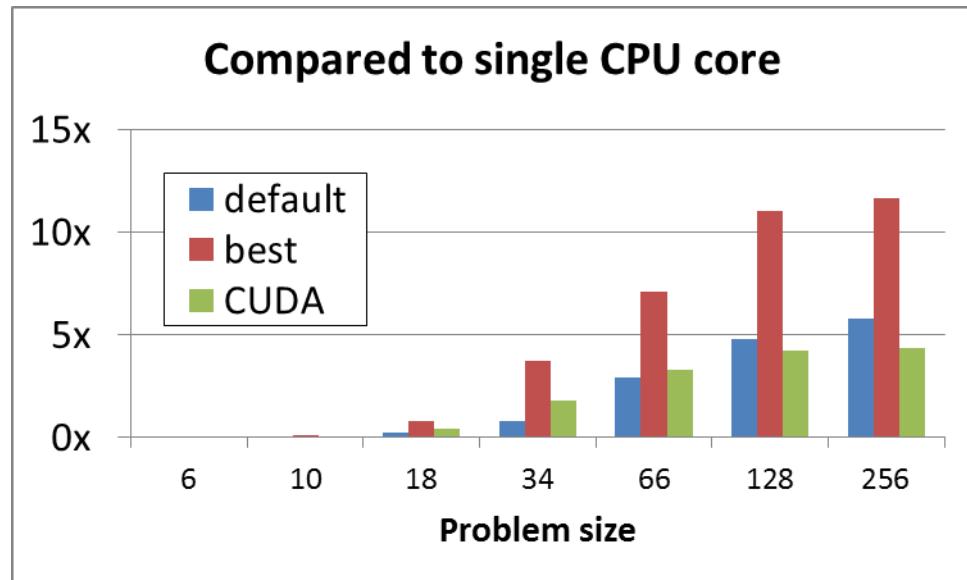
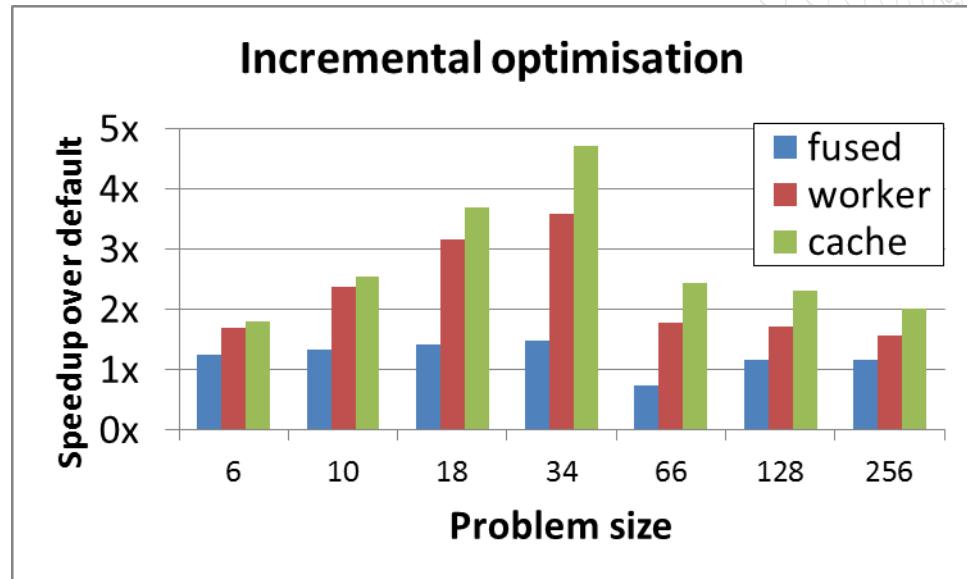
- best algorithm choice was:
 - single loopnest, worker clause, 1d cache
 - vector_length either 256 or 512 (similar)
 - did not investigate tiling
- This was best for all problem sizes
 - so we can use same algorithm

```
!$acc parallel loop
DO k = 2,n3-1
 !$acc loop worker
   DO j = 2,n2-1
     DO i = 2,n1-1
       !$acc cache( u(i-1:i+1,j,k) )
         r(i,j,k) = <full stencil on u>
     ENDDO
   ENDDO
 ENDDO
```

Comparing performance

- Tuning can work well:
 - 1.6x to 2.0x gained here
 - depending on problem size

- OpenACC vs. CPU core:
 - GPU wins for larger problems
 - default: for 34 and larger
 - tuned: for 18 and larger
- OpenACC vs. CUDA
 - even default generally better
 - CUDA here relatively untuned:
 - fused, collapsed, no __shared__
 - but it's much easier to tune OpenACC



Other Performance Tuning Improvements

- **Avoiding temporary arrays in resid:**
 - benefit v. small
 - was it really worth hacking the source?
- **Call an external CUDA version of resid**
 - First attempt (no use of `__shared__`) is slower
 - This was a naive kernel (there is a lesson in this...)

	Original (Mop/s)	Data Region (Mop/s)	Code change (Mop/s)	CUDA kernel (Mop/s)
Fortran	4787	27583	28993	24262
C	4849	26476	30718	23794

- **Optimising data movements is much more important than kernel tuning**

COMPUTE

STORE

ANALYZE

Conclusions: How far did we get?

- Significant speedup compared to single core:
 - Mop/s: $4800 \rightarrow 27000 = 5\frac{1}{2}x$
- The real comparison is to a full CPU
 - run the OpenMP version of the code on the Intel SandyBridge CPU
 - 4 threads optimal: gives 15000 Mop/s
 - maybe MPI version would scale better, but our code here is scalar
- Final speedup compared to full CPU: **1.8x**

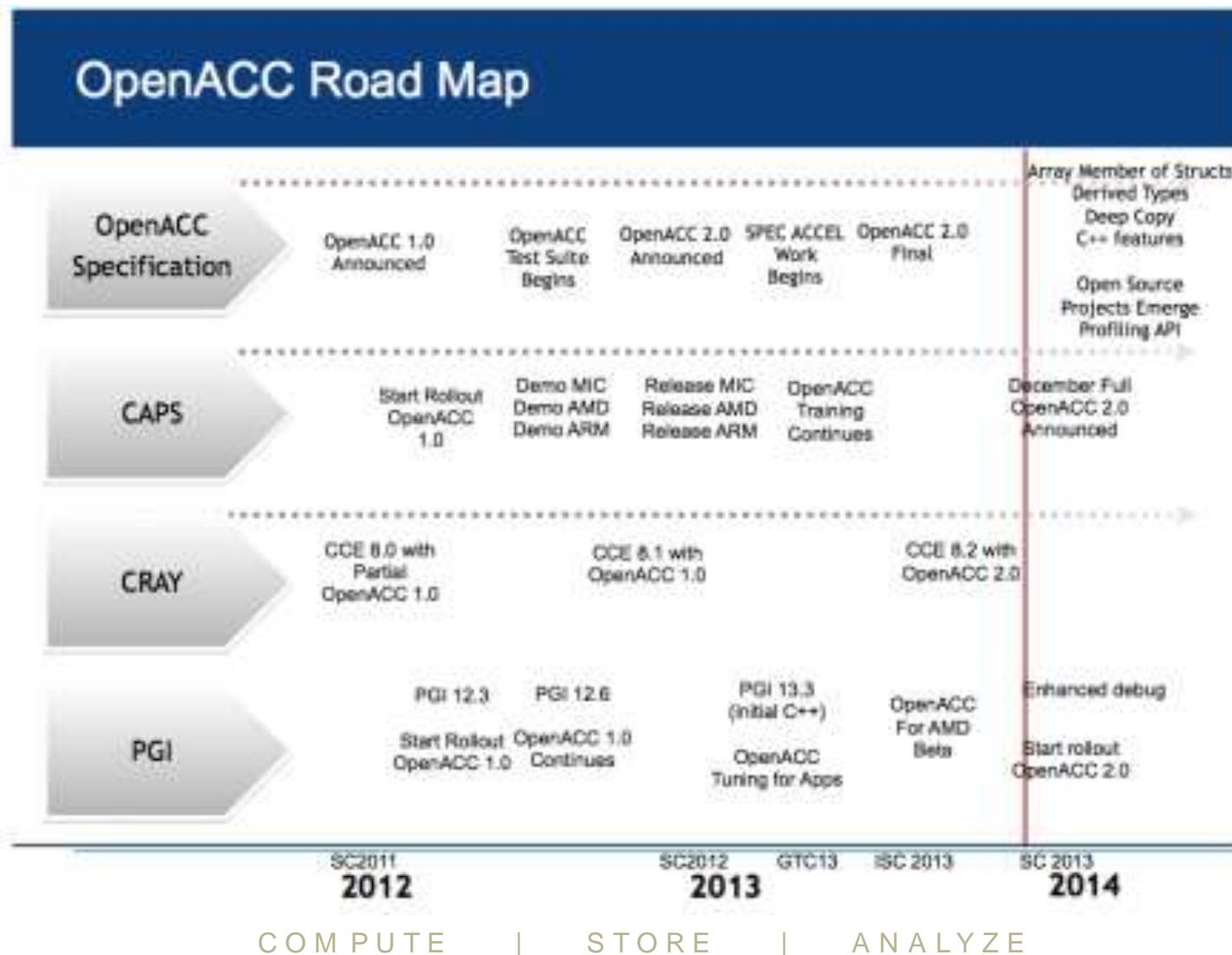
Conclusions: How much further could we get?

- So we are **1.8x faster, socket-for-socket**
 - How much faster could we get (speeds and feeds)
 - Flops and memory b/w around 5x faster than single Intel Sandybridge
- So why were we not faster?
 - MultiGrid application cycles through grid sizes
 - sometimes the grid is really small: 4x4x4
 - CrayPAT loop profiling showed us that
 - Small grid sizes will never schedule well on the GPU
 - consider checking grid size and using different OpenACC for different sizes
 - or do we even need the smaller grids?

OpenACC 6: predicting the future

Alistair Hart
Cray Exascale Research Initiative Europe

The roadmap so far



OpenACC 2.1 and beyond

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

Flat object model

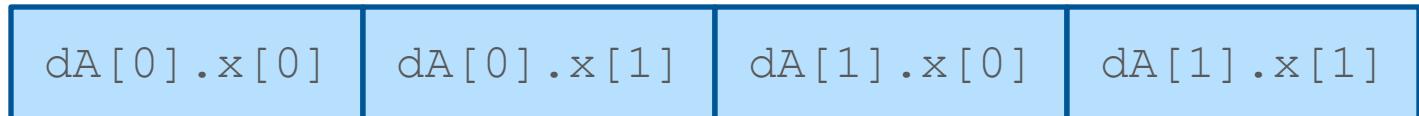
- OpenACC supports a “flat” object model
 - Primitive types
 - Composite types without allocatable/pointer members

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

Host Memory:



Device Memory

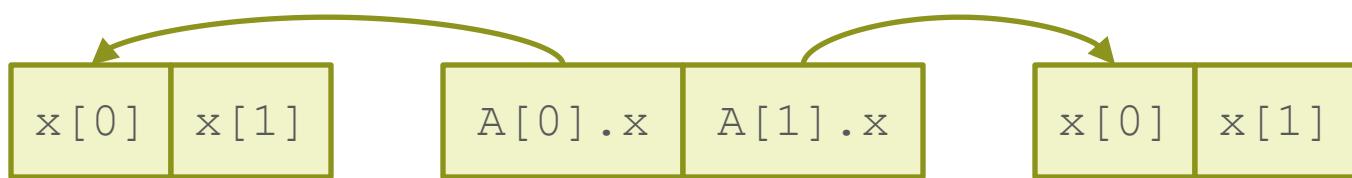


Challenges with pointer indirection

- Non-contiguous transfers
- Pointer translation

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

Host Memory:

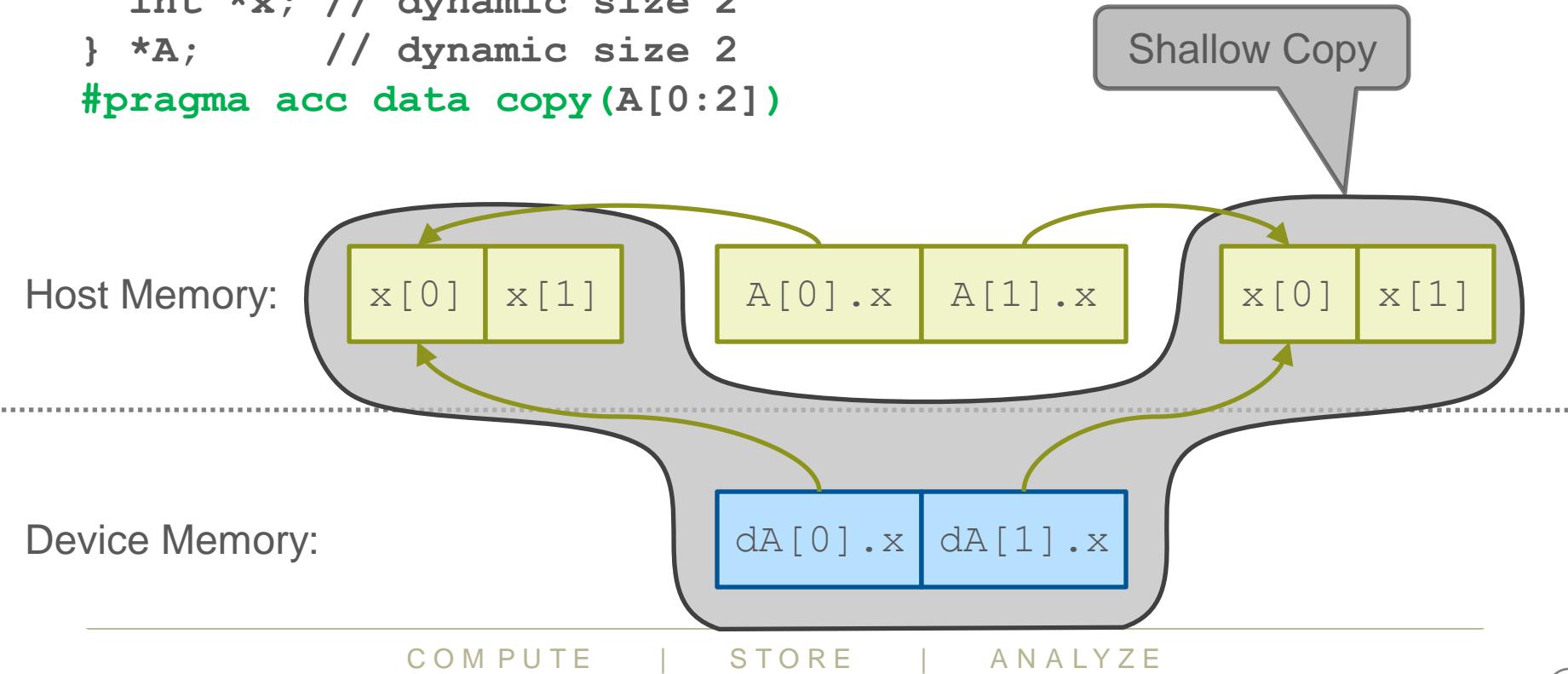


Challenges with pointer indirection

- Non-contiguous transfers
- Pointer translation

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

Host Memory:



Challenges with pointer indirection

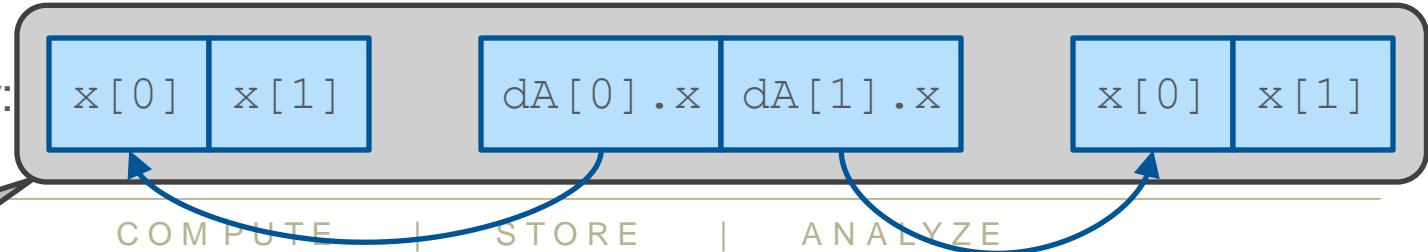
- Non-contiguous transfers
- Pointer translation

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

Host Memory:



Device Memory:



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

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

Proposed “member shape” directives

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

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

Member-shape directive examples

```
extern int size_z();  
int size_y;  
struct Foo  
{  
    double* x;  
    double* y;  
    double* z;  
    int    size_x;  
    // deep copy x, y, and z  
    #pragma acc declare shape(x[0:size_x], y[1:size_y-1], z[0:size_z()])  
  
type Foo  
    real,allocatable :: x(:)  
    real,pointer      :: y(:)  
    !$acc declare shape(x)    ! deep copy x  
    !$acc declare unshape(y) ! do not deep copy y  
end type Foo
```

OpenMP 4.0

OpenMP® accelerator directives

- Announced 15yrs ago
- Works with Fortran, C, C++
- An established open standard is the most attractive
 - portability; multiple compilers for debugging; permanence
- **Last version 3.1 (July 2011)**
 - A common directive programming model for shared memory systems
 - 354 pages
- **Latest version 4.0 (July 2013)**
 - A common programming model for wide range of accelerators
 - 320 pages
 - 140 pages moved to an examples document!
- **OpenACC will continue to be supported**
 - Developers can transition to OpenMP if they wish
 - Converting OpenACC to OpenMP will be straightforward
 - More on this later

OpenMP 4.0 “device” additions

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

OpenACC compared to OpenMP

OpenACC

- **Parallel (offload)**
 - Parallel (multiple “threads”)
- **Kernels**
- **Data**
- **Loop**
- **Host data**
- **Cache**
- **Update**
- **Wait**
- **Declare**

OpenMP

- **Target**
- **Team/Parallel**
-
- **Target Data**
- **Distribute/Do/for**
-
-
- **Target Update**
-
- **Declare target**

OpenACC compared to OpenMP continued

OpenACC

- enter data
- exit data
- data api
- routine
- async wait
- parallel in parallel
- Tile
- Device_type

OpenMP

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

OpenACC compared to OpenMP continued

OpenACC

- atomic
 -
 -
 -
 -
 -
 -
 -
 -
 -

OpenMP

- Atomic
 - Critical sections
 - Master
 - Single
 - Tasks
 - barrier
 - `get_thread_num`
 - `get_num_threads`
 - `WU`

OpenMP async

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

Transition from OpenACC to OpenMP

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

Sources of further information

- **Standards web pages:**

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

- **Discussion lists:**

- Cray users: openacc-users@cray.com
 - automatic subscription if you have a swan (or raven) account
- Fora: openacc.org/forum , openmp.org/forum

- **CCE man pages (with `PrgEnv-cray` loaded):**

- programming model and Cray extensions: `intro_openacc`
- examples of use: `openacc.examples`
- also compiler-specific man pages: `crayftn`, `craycc`, `crayCC`

- **CrayPAT man pages (with `perftools` loaded):**

- `intro_crayspat`, `pat_build`, `pat_report`
 - also command: `pat_help`
- `accpc` (for accelerator performance counters)

!\$acc exit data