Directive-based programming for GPUs, accelerators and HPC

Alistair Hart, Harvey Richardson (Cray ERI)
Alan Gray (EPCC ETC),
Karthee Sivalingham (EPCC & University of Edinburgh)

email: ahart@cray.com
Cray Exascale Research Initiative in Europe

- Launched December 2009
  - Initial research partners are EPCC and CSCS
  - Cray increased Edinburgh-based team with Alistair Hart and Harvey Richardson
- Exploring how real applications can exploit future Exascale architectures, specifically:
  - Programming models for PGAS languages
  - Programming GPU accelerators
  - Improved algorithms for FFTs
  - Network and I/O profiling
- Strong interactions with Cray R&D
Performance milestones towards exascale

1 GF sustained – 1988: Cray Y-MP; 8 Processors
- Static finite element analysis

1 TF sustained – 1998: Cray T3E; 1024 Processors
- Modeling of metallic magnet atoms

1 PF sustained – 2008: Cray XT5; 150k Processors
- Superconductive materials

1 EF sustained – ~2018: Cray *; ~10M Processors

(Barry Bolding, Cray)
Exascale needs heterogeneous computing

• Scale and sustained performance per \(\$,\text{watt}\) requires:
  ▪ heterogeneous node architecture
  ▪ deep, explicitly managed memory hierarchy
  ▪ microarchitecture to exploit parallelism at all levels of code

• This closely resembles today's GPU accelerator...
  ▪ GPUs will feature in next generation supercomputers
  ▪ longer term, hardware will evolve, but GPUs still useful for exploring programming models and challenges

• The real challenge: exploiting enough parallelism in applications
  ▪ Identifying parallelism is hard
  ▪ Mapping parallelism to hardware is much easier
  ▪ Accelerator directives are designed to help
Accelerator programming

• There are already many ways:
  ▪ CUDA (incl. PGI CUDA Fortran), Stream, OpenCL, hiCUDA
  ▪ All are quite low-level and closely coupled to the GPU

• User needs to write specialist kernels:
  ▪ Hard to write and debug
  ▪ Hard to optimise for specific GPU architecture
  ▪ Hard to update (porting/functionality)

• Accelerator directives provide higher-level approach
  ▪ Based on original source code
    ▪ Easier to maintain/port/extend code
    ▪ Users with OpenMP experience find it a familiar programming model
  ▪ Possible performance sacrifice
    ▪ Important to quantify this (and tune the compiler)
OpenMP accelerator directives

- An open standard is the most attractive for developers
  - portability; multiple compilers for debugging; permanence
- Subcommittee of OpenMP ARB, aiming for OpenMP 4 (2011?)
  - includes most major vendors + others (e.g. EPCC)
  - co-chaired by Cray (James Beyer)
- Cray is an enthusiastic supporter
  - CCE is first full implementation
    - now performance tuning
    - Fortran, C, C++
    - Draft standard; Will track standard as it evolves
    - compiles straight to PTX
    - Alpha release now available
      - contact ahart@cray.com to request access

- Expect other vendors to follow lead
OpenMP accelerator directives

- Modify original source code with directives
  - Non-executable statements (comments, pragmas)
    - Ignored by non-accelerating compiler
  - `omp` sentinel (like original OpenMP)
  - Accelerator constructs start with `acc_`
  - Fortran:
    - Usually paired with `!$omp end acc_*`
  - C/C++:
    - Structured block {...} avoids need for end directives
  - Some preprocessing when mixing `!$omp` with `!$omp acc_`
    - e.g. when using non-accelerating compiler
    - a compiler flag (something like `–h noomp_acc`) would be nice
    - Otherwise, mild use of the preprocessor
      - Accelerating compiler provides: `#define _OPENMP_ACCEL`
A first example

Execute a loop nest on the GPU

- Data movement
  - Compiler does this
    - copies $a(2:N,1:M)$, $b(2:N,1:M)$ to GPU at start of region
    - copies $c(2:N,1:M)$ back to CPU at end of region
  - correct, but not necessarily optimal

- Loop schedule: spreading loop iterations over PEs of GPU
  - i.e. divided between SIMD and MIMD blocks of GPU
  - Compiler chooses appropriate schedule for loop(s)
  - Not always optimal yet
    - performance modelling very hard for general code

- Tune default behaviour with optional clauses on directives

```c
!$omp acc_region
!$omp acc_loop
DO j = 1,M
  DO i = 2,N
    c(i,j) = a(i,j) + b(i,j)
  ENDDO
ENDDO
!$omp end acc_loop
!$omp end acc_region
```
Clauses for acc_region

• Use these first to begin to tune the accelerator kernels
• Data clauses:
  ▪ **acc_copy, acc_copyin, acc_copyout**
    • copy moves data "in" to GPU at start of region and "out" to CPU at end
    • supply list of arrays or array sections (using Fortran ":" notation)
  ▪ **acc_shared**
    • Shared by all threads on accelerator
    • No copyin/out – useful for temporary arrays in loopnests
  ▪ **private, firstprivate**
    • As OpenMP. Loop variables and scalars are automatically private
  ▪ **default(<any of above>|none)**
  ▪ **present**: described presently
More clauses for acc_region

- Other clauses:
  - **if**(scalar-logical-expression)
    - Executes on host if scalar-logical-expression evaluates to .FALSE. at runtime
    - Useful if small problem sizes are not worth sending to GPU
      - Can result in extra copying inside data region
  - **device**(integer-expression)
    - Choose which of accelerators present to use
  - **num_pes**(depth:num [, depth:num])
    - How many PEs to use at each level (e.g. SIMD, MIMD)
  - **async**(handle)
    - Launch accelerator region asynchronously
    - Allows overlap of CPU and GPU computation
    - handle + runtime functions to wait/test for completeness
Clauses for acc_loop

- Some similar to OpenMP:
  - reduction(operator:list)
  - collapse(n): collapse outer n loops into single loop

- Most associated with performance tuning
  - **schedule**, **level**
    - specifies the level of parallelism to use (SIMD, MIMD etc.)
    - max_par_level, num_pes used to further tune schedule
  - **cache**(obj[:depth])
    - choose where to put data in GPU memory hierarchy
  - **seq**: this loop of loop nest to execute sequentially
    - **kernel**: this *and all contained* loops to execute sequentially
  - **vector**: recurrance/data dependency requires vectorisation

- **hetero**: provides manual load-balancing
  - divide loop iterations between GPU and CPU in specified way
 !$omp acc_region_loop

- Shortcut for accelerating single loop nest
  - combines acc_region with acc_loop
    - clauses as for acc_region and acc_loop
  - saves a full two lines of source code (!), BUT...
  - ... v. important for implementing hetero clause for acc_loop
Data regions

- Why do we need !$omp acc_data $? 
  - `acc_region` can span multiple `acc_loop`'s 
    - data movement only occurs at start and end of region 
  - Sometimes want to share data across multiple `acc_region`'s 
    - e.g. if want to write to STDOUT, disk between regions 
    - e.g. if you want to time the kernels 

- Using `acc_data` 
  - Defines "explicit" data region spanning multiple `acc_region`'s 
  - data movement only at start/end of `acc_data` region 
    - User can use `acc_update` (see later) to move data within region 

- Clauses similar to `acc_region` data clauses
Implicit data regions

- Directive `acc_data` defines an explicit data region
- An implicit data region:
  - entire programming unit where object exists
    - global objects: entire program
    - local objects: current programming unit and its call tree
  - More powerful than `acc_data`
- Declarative directives apply to implicit data region:
  - `!$omp acc_res(list)`
    - Objects exist only on accelerator
      - implicit data region equivalent of `!$omp acc_data acc_shared(list)`
      - Can be ALLOCATABLE arrays
  - `!$omp acc_mirror(list)`
    - Object exists in both CPU and GPU memory spaces
    - mirrors allocation status, NOT actual data (use `acc_update` for that)
Other constructs

• Copy between memory spaces inside data region:
  ▪  !$omp acc_update
    • clause acc(obj) copies obj from CPU to GPU
    • clause host(obj) copies obj from GPU to CPU
    • (obj1:obj2) uses obj2 as the target for the copy
    • can update full arrays or just array sections or elements

• Barriers for synchronisation
  ▪  !$omp acc_barrier(lvl)
    • Ensures all PEs have reached this point
    • lvl determines level in execution hierarchy to synchronise
    • NVIDIA: lvl=1 between SIMD threadblocks, lvl=2 within SIMD threadblock
      • end acc_region acts as lvl=1 barrier
Sharing data between subprograms

• If your code is simple, previous directives are all you need
  • "simple" means:
    • Profile dominated by a few subprograms
    • In these, data movement far outweighed by computation
    • Subprograms do not call other subprograms (or you can easily inline or refactor to arrange this)
  • Examples of this?
    • WRF (see PGI website)
    • HPsrc
• If not, would like to be able to hold data on accelerator as we move between subprograms
  • !$omp acc_present, !$omp acc_call* constructs allow this
Sharing GPU data between subprograms... 1

**double_me(a)** is a call to a *host* kernel

- **a** exists on CPU & GPU; data region has **acc_copy** statement
  - **b** will not be copied to/from GPU at start/end of subprogram **double_me()**
- **acc_region** checks if **b** already on GPU:
  - yes: it uses this without copies; no: it follows the **acc_copy(b)** clause
Sharing GPU data between subprograms... 2

• double_me(a) is a call to an accelerator kernel
  ▪ b is explicitly defined to be present on the accelerator
    • b will not be copied to/from GPU at start/end of subprogram double_me()
  ▪ acc_region is now implicit, but acc_loop is still needed

PROGRAM main
  USE stuff
  INTEGER :: a(N)
!$omp acc_res(a)

!$omp acc_region_loop
  a(:) = [ (i,i=1,N) ]
!$omp end acc_region_loop

!$omp acc_call present(a)
  CALL double_me(a)
END PROGRAM main

MODULE stuff
  CONTAINS
  !$omp acc_call_definition present(b)
  SUBROUTINE double_me(b)
    INTEGER :: b(:)
    !$omp acc_loop
      b(:) = 2*b(:)
    !$omp end acc_loop
  END SUBROUTINE double_me
END MODULE stuff
More succinctly... 3

• Only need `!$omp_acc_call_declaration` once per subprogram
  ▪ Replaces:
    • `!$omp acc_call_definition` at top of subprogram
    • `!$omp acc_call` at every call site
  ▪ Well, once per interface block (Fortran) or header (C/C++)

```plaintext
PROGRAM main
  USE stuff
  INTEGER :: a(N)
!$omp acc_res(a)

!$omp acc_region_loop
  a(:) = [ (i,i=1,N) ]
!$omp end acc_region_loop

  CALL double_me(a)
END PROGRAM main
```

```plaintext
MODULE stuff
  CONTAINS
!$omp acc_call_declaration present(b)
  SUBROUTINE double_me(b)
    INTEGER :: b(:)

!$omp acc_loop
  b(:) = 2*b(:)
!$omp end acc_loop

END SUBROUTINE double_me
END MODULE stuff
```
 !$omp acc_call* clauses

- **async**(handle): calls to this kernel are asynchronous
  - Use the handle to test/wait for completeness
- **type**(acc|host|both)
  - create code versions for accelerator, host or both
- **if**: provides runtime choice to run on accelerator or host
- **acc_copy**: tunes data movement at start/end of kernel
- **present**
  - specifies that objects are already on the accelerator
  - key to keeping data on the accelerator between subroutine calls
PGI directives

- A very similar model to OpenMP
  - not surprising – it acted as a template for first draft
- Supported by PGI compiler
  - Fortran and C (not C++)
  - Some limitations in C support
    - some in implementation, some in design

- Basic differences (sentinel, directive names)
- Compiler feedback very useful in tuning kernels
- Other directives-like approaches
  - Intel: `#pragma offload target(MIC), ...`
  - We must avoid a Tower of Babel, hence OpenMP efforts
Some examples

• Aims:
  ▪ Investigate how easy it is to correctly accelerate codes
  ▪ Understand performance barriers
    • and how to overcome them
  ▪ Compare the performance with CUDA and/or CPU
  ▪ Guide next steps in directive model/implementation

• Accelerate and optimise three sample applications:
  1. NAS Parallel Benchmarks Multigrid code
  2. HPsrc
  3. Ludwig
NPB MG description

- NAS Parallel Benchmarks (version 3.3.1)
  - Used to test a lot of parallel programming paradigms
  - OpenMP version: used as template
- MG (multigrid) solves Laplacian on 3D grid
  - Fortran77 (with homages to earlier versions)
  - Three main hotspot subroutines:
    - resid (50% of runtime)
    - psinv (25%)
    - rprj3 (9%)
  - Data arrays passed to/from subroutines at every iteration
NPB MG acceleration

- Accelerated with OpenMP accelerator directives
  - using CCE v7.4
  - supports data sharing
- No performance tuning yet
  - Concentrated on correct functionality
  - PGI and CCE untuned versions perform very comparably
    - estimate kernels using PGI timer
- Some workarounds needed for PGI
  - mainly explicit privatisation of arrays
    - support from PGI forum was excellent
  - CCE worked "out of the box"
    - once the non-standard Fortran had been fixed
NPB MG performance (class A, 256 x 256 x 256)

NPB-OMP-ACC/MG

- CPU
- GPU
- acc_present

NPB MG performance (class A, 256 x 256 x 256)
PGI directives show v. similar pattern

NPB-PGI/MG

- CPU
- just kernels
HPsrc

• The HPsrc code was developed for lattice QCD

• Karthee Sivalingham described this work yesterday

• PGI directives:
  ▪ 52x speed-up over single-thread CPU (original)
    ▪ tuning increased performance 11-fold
  ▪ Directive performance penalty very acceptable (~20%)
    ▪ Matches best single-level CUDA performance
    ▪ CUDA code more difficult to convert to two-level parallelism

• CPU performance:
  ▪ Best Fortran version for GPU runs 20% slower on CPU
Best HPsrc performance vs. problem size

Speed, relative to single level CUDA

- CUDA
- PGI
- CPU
- PGI two-level

npoints

0 4096 8192 12288 16384 20480 24576
Ludwig description

• Simulates hydrodynamics of complex fluids
  ▪ Regular 3d grid; collision kernel takes 90% of runtime

• Accelerated and optimised using PGI directives
  ▪ Temporary scalars to force register use
    • Sped up the CPU version 2-3x (PGI; less so for CCE)
  ▪ Manual inlining needed to avoid accelerator function calls
  ▪ Workarounds for PGI accelerator support for C structs
    • Temporary arrays instead of structs
    • Use array pointers, assuming structs laid out contiguously
  ▪ Performance limited by only single-level parallelism
    • Issue with code structure and also with PGI accelerator and C
    • User must explicitly privatise to place extra index correctly for coalescing
Ludwig performance plot

PGI Acceleration of Ludwig Collision Kernel
128³, 100 Iterations

Data transfers can be reduced

Kernel 8x faster
Conclusions of case studies

The optimisation procedure:
1. Optimise data movement
2. Refactor code to expose as much parallelism as possible
   - compiler feedback has to be read carefully
3. Make sure array accesses are coalescing
   - especially with single-level parallelism on outer loop
4. Privatise arrays explicitly for optimal index placement
   - trial and error where this is not obvious
5. Hold data on GPU where possible
6. Optimised single-source code for both CPU and GPU difficult
   - Architectures very different
   - Architecture-dependent source is not a new problem
Summary

• Introduced accelerator directives: OpenMP
  ▪ Attractive programming model
  ▪ Based on original Fortran, C and C++ codes
• We can now share data between subprograms (CCE)
  ▪ Suitable for much wider class of applications now
    • demonstrated in MG code
• Directives are easier to tune
  ▪ Tuning boosted PGI kernel from 5x → 50x CPU performance
  ▪ “Performance penalty” for directives can be small
    • Got 80% of equivalent CUDA performance
  ▪ Refactoring for two-level parallelism much easier
    • Boosted performance further, especially for small problem sizes
Helping users exploit GPUs

- Users need good compilers, good feedback and good training
  - How do we make it easier for application developers?
- Optimisation steps are counter-intuitive for many users
  - Split p-loop vs. “long loops optimise cache usage”
  - Make p the fastest-moving array index vs. “access memory sequentially”
  - It’s back to vectorisation again
  - How do we write code that runs well across heterogeneous architecture?
- Cray developing full Programming Environment
  - Tools to expose and exploit every bit of parallelism in their code
    - e.g. Craypat now does loop trip-count statistics
  - Integrated performance analysis for GPU and CPU
    - Need to understand full application
  - 30 years experience of optimised vectorisation in the CCE
  - Optimised GPU libraries (BLAS, FFT...)
Acknowledgments

Thank you to those that helped us get to grips with directives:

- EPCC Exascale team
- PGI
  - Doug Miles and Mathew Colgrove
- Cray R&D team
  - John Levesque, Luiz DeRose, James Beyer, David Oehmke...

and you may have seen our poster at SC10...