

Accelerator directives: A user's perspective

Alistair Hart, Harvey Richardson (Cray ERI)

Alan Gray (EPCC ETC), Karthee Sivalingham (EPCC)

Cray Exascale Research Initiative and EPCC

- Launched December 2009; based in Edinburgh
 - Joint with EPCC's Exascale Technology Centre
 - Builds on Cray HECToR Centre of Excellence
- Exploring how real applications can exploit exascale machines, specifically:
 - Programming models for PGAS languages
 - Programming GPU accelerators
 - Improved algorithms for FFTs
 - Network and I/O profiling
- Strong interactions with Cray R&D
- Parallel project with CSCS (Lugano, Switzerland)



Contents

- Exascale computers
 - When will we have one?
 - What will it look like?
 - How will we program them?
- Accelerator directives:
 - Why do we need them?
 - How do we use them?
- Performance tuning examples
 - Case study in directive-based optimisation on GPU
- Summary
 - Including upcoming Cray training courses

Performance milestones towards exascale

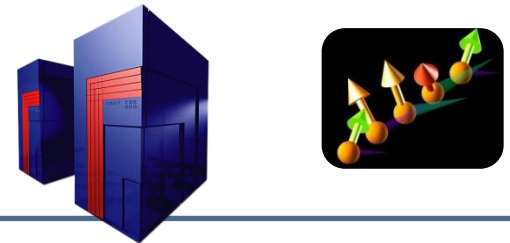
1 GF sustained – 1988: Cray Y-MP; 8 Processors

- 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



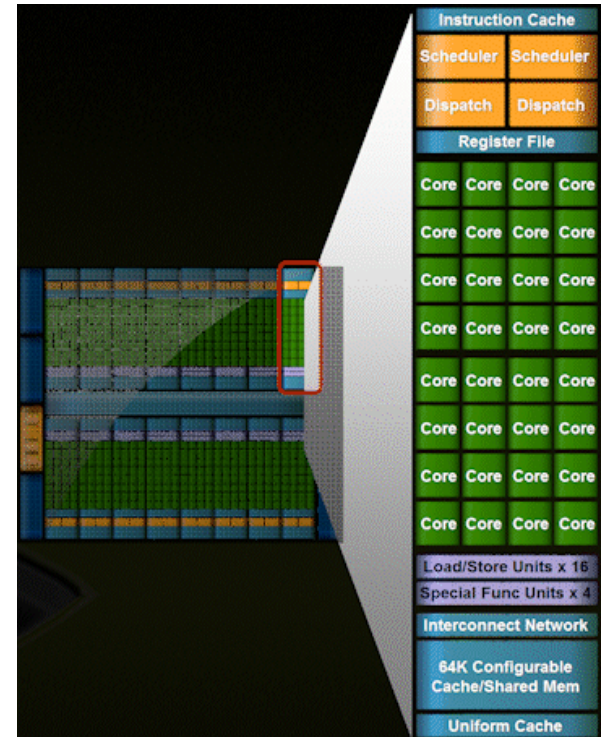
1 EF sustained – ~2018: Cray *; ~10M Processors

Exascale needs heterogeneous computing

- Every 10 years: 1000x performance, 100x cores
- Scale and sustained performance per {\$,watt} requires:
 - heterogeneous node architecture
 - deep, explicitly managed memory hierarchy
 - microarchitecture to exploit parallelism at all levels of code
- This sounds a lot like a GPU accelerator...
- The real challenge: exploiting enough parallelism in applications
 - Mapping parallelism to hardware is then much easier
 - Cray is developing full PE to support hybrid XE6-Fermi blades
 - Compilers and static analysis
 - Optimised GPU libraries
 - Runtime performance and analysis tools
 - Accelerator directives are an important part of this

Accelerators

- Term covers a range of different architectures
 - GPUs (NVIDIA, AMD...)
 - Cell
 - Embedded and FPGAs
 - Clearspeed
 - Intel (Knights Ferry etc.)
- Common features
 - Highly parallel
 - Vector architecture, e.g. NVIDIA:
 - Cores divided into “SM” groups (256-way)
 - “Threads” operate in SIMD blocks
 - “Coalescing”: best performance when neighbouring threads load consecutive memory addresses
 - Usually a separate memory space (likely to evolve)



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 (see later results)

PGI directives

- PGI: <http://www.pgroup.com/resources/accel.htm>
 - Fortran: **!\$acc;** C: **#pragma acc**
- Basic use:
 - simply surround parallel region:
 - **!\$acc do** optional (unlike OpenMP)
- **pgfortran -ta=nvidia -Minfo=accel program.f03**
 - Produces accelerated kernels with correct data movement
 - Compiler feedback very important for tuning (see later)
 - Currently targets only NVIDIA GPUs

```
!$acc region  
<loop nest>  
!$acc end region
```

PGI tuning directives

Optional additional directives to tune performance:

- **!\$acc region [clause(s)]**
 - **copy/copyin/copyout/local**: tune data movement
 - **if**: runtime control of where to run kernel: GPU or CPU
- **!\$acc do [clause(s)]**
 - **private**: privatise internal arrays
 - **parallel, vector, ...**: tune the loop schedule
 - **cache**: optimise memory usage
- **!\$acc data region**
 - Share data between kernels
 - Currently just in same subprogram



OpenMP draft directives

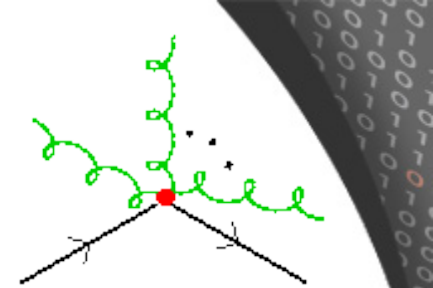
- Aims to be part of OpenMP 4.0 standard
 - Subcommittee (Cray, PGI, CAPS, ...) – co-chaired by Cray
 - Fortran: **!\$omp acc_**
 - C/C++: **#pragma omp acc_**
- Used in similar way to PGI directives:
- Tuning clauses include:
 - **!\$omp acc_region** clauses for data movement
 - **!\$omp acc_loop** clauses for loop scheduling, caching...
- Supported in Cray Compilation Environment (CCE)
 - Internally already, release provisionally Q4 2010.
 - Also currently targets only NVIDIA GPUs.

```
!$omp acc_region
!$omp acc_loop
<loop nest>
!$omp end acc_loop
!$omp end acc_region
```

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
 - Pinpoint what is needed next
- Accelerate and optimise three sample applications:
 1. HPsrc
 2. Ludwig
 3. NAS Parallel Benchmarks Multigrid code

HPsrc



- The HPsrc code was developed for lattice QCD
 - Performs low-dimensional integrations of double-precision complex-valued functions: [Comput. Phys. Commun. 180 \(2009\) 2698](#).
- Very GPU friendly:
 - Lots of loop-based parallelism
 - Very little data movement per GPU flop.
- Case study for state-of-the-art accelerator directives compilers:
 - Profile dominated by single kernel
 - Around 80% of runtime on weighted sums of double-complex EXPs
- Compare performance against hand-written CUDA-C kernel
 - Concentrate here on PGI compiler
 - CCE version working (no issues with complex intrinsics)

Very simplified HPsrc: original kernel with PGI

Avoids complex workarounds

Typical value

```
INTEGER, PARAMETER : Nterms = 8000
REAL(kind=r1_kind), INTENT(in)  :: k(0:3,Order,npoints)
INTEGER,                INTENT(in)  :: yxv(0:3,Order,Nterms)
REAL(kind=r1_kind), INTENT(in)  :: f(Nterms)
REAL(kind=r1_kind), INTENT(out) :: vtx(npoints)
```

Typically thousands

```
!$acc region
```

```
!$acc do private(p3)
```

Compiler tells you to add this

```
DO p = 1,Npoints
```

```
  DO i = 1,Nterms
```

```
    phase3 = 0.5d0 * SUM(k(:, :, p) * yxv(:, :, i))
```

```
    p3(i) = f(i) * EXP(phase3)
```

```
  ENDDO
```

```
  vtx(p) = SUM(p3)
```

```
ENDDO
```

```
!$acc end region
```

Optimising data movement

```
REAL(kind=r1_kind), INTENT(in) :: k(0:3,Order,npoints)
INTEGER, INTENT(in) :: yxv(0:3,Order,Nterms)
REAL(kind=r1_kind), INTENT(in) :: f(Nterms)
REAL(kind=r1_kind), INTENT(out) :: vtx(npoints)
```

```
Generating copyin(k(0:ndir-1,1:nlegs,1:npoints))
Generating copyin(yxv(0:ndir-1,1:nlegs,1:nterms))
Generating copyin(f(1:nterms))
Generating copyout(vtx(1:npoints))
```

- Check if the data movement is optimal (not just correct)
 - Directions: eliminate unnecessary copyin/out
 - e.g. if compiler wasn't sure it was a temporary array
 - Contiguous array sections transfer fastest
 - Compiler only transfers minimum number of bytes
 - Both fine in this example

Loop scheduling

```
DO p = 1,Npoints
  DO i = 1,Nterms
    phase3 = 0.5d0 * SUM(k(:, :, p) * yxv(:, :, i))
    p3(i) = f(i) * EXP(phase3)
  ENDDO
  vtx(p) = SUM(p3)
ENDDO
```

```
302, Loop is parallelizable
Accelerator kernel generated
!$acc do parallel, vector(256)
CC 2.0 : 0 registers; 0 shared, 0 constant, \\  
0 local memory bytes; 100 occupancy
303, Loop is parallelizable
307, Loop is parallelizable
329, Loop is parallelizable
```

Parallelizable ≠ Parallelised!

Initial performance

- Currently only parallelising outer p-loop
- Wrote equivalent CUDA version
- Compared CPU with PGI and CUDA on Fermi C2050 GPU

Kernel	CPU	PGI directives	CUDA
Time (ms)	3102	656	336

(Npoints=4096)

- Disappointing:
 - Less than 5x faster than *single-thread* on CPU
 - More than 2x slower than CUDA
- Single-level parallelism compromises performance
 - Options to proceed (depends on code):
 1. Squeeze best performance from single-level parallelism
 2. Refactor code to exploit parallelism of inner i-loop

Option 1: optimise single-level parallelism

- Reorder $k(p, :, :)$ for coalescing
 - Data rearrangement times negligible
 - Same change to the CUDA version
- Explicitly privatise $p3(p, i)$ for coalescing
 - private clause generated suboptimal $p3(i, p)$
 - All these changes: PGI compiler feedback same

```
!$acc region local(p3)
```

```
DO p = 1, npoints
```

```
DO i = 1, Nterms
```

```
phase3 = 0.5d0 * SUM(k(p, :, :) * yxv(:, :, i))
```

```
p3(p, i) = f(i) * EXP(phase3)
```

```
ENDDO
```

```
vtx(p) = SUM(p3(p, :))
```

```
ENDDO
```

```
!$acc end region
```

No copyin/out

Option 1: optimise single-level parallelism

- Compare performance:

Version	CPU	PGI directives	CUDA
original	3102	656	336
single-level	4680	83	61

- PGI directives:
 - 37x speed-up over single-thread CPU (original)
 - 8x speed-up by refactoring code
- CPU performance:
 - Best Fortran version for GPU runs 50% slower
- But we can do better...

Option 2: refactor for two-level parallelism

- Split p-loop so loop nests are “full depth” throughout
 - Forced to explicitly privatise p3
 - First loop wants $p3(i,p)$; second loop wants $p3(p,i)$
 - Trial and error: add p as fastest index for coalescing

```
!$acc region local(p3)
DO p = 1,npoints
  DO i = 1,Nterms
    phase3 = 0.5d0 * SUM(k(:, :, p) * yxv(:, :, i))
    p3(p, i) = f(i) * EXP(phase3)
  ENDDO
ENDDO
DO p = 1,npoints
  vtx(p) = SUM(p3(p, :))
ENDDO
!$acc end region
```

Option 2: refactor for two-level parallelism

```
DO p = 1,npoints
  DO i = 1,Nterms
    phase3 = 0.5d0 * SUM(k(:, :, p) * yxv(:, :, i))
    p3(p, i) = f(i) * EXP(phase3)
  ENDDO
ENDDO
DO p = 1,npoints
  vtx(p) = SUM(p3(p, :))
ENDDO
```

Accelerator kernel generated

```
302, !$acc do parallel, vector(16)
```

```
303, !$acc do parallel, vector(16)
```

 Cached references to size [16] block of 'f'

Accelerator kernel generated

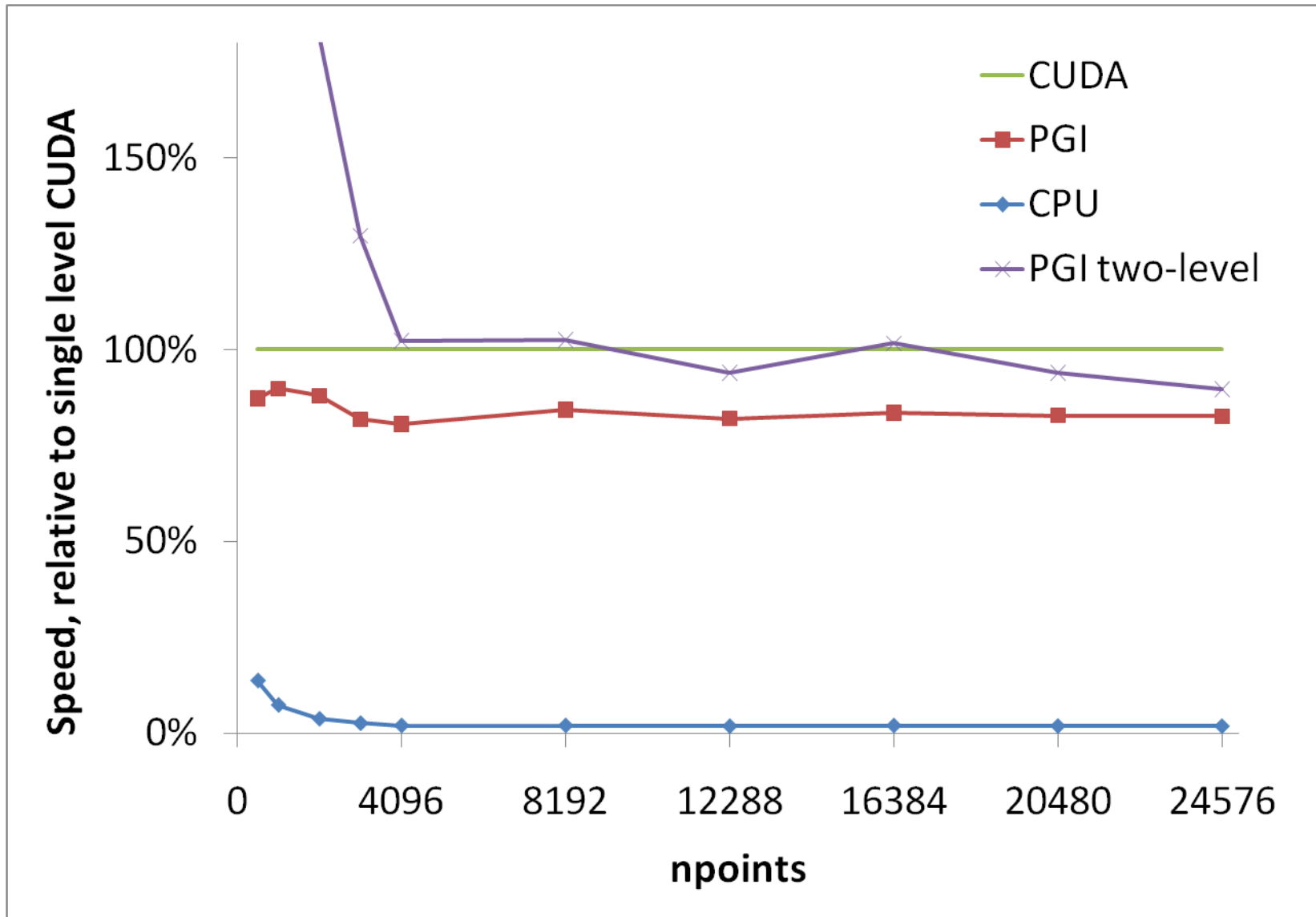
```
321, !$acc do parallel, vector(256)
```

Option 2: refactor for two-level parallelism

Version	CPU	PGI directives	CUDA
original	3102	656	336
single-level	4680	83	61
two-level	3979	60	-

- PGI directives:
 - 52x speed-up over single-thread CPU (original)
 - 11x speed-up by refactoring code
 - 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

Best HPsrc performance vs. problem size



Conclusions of HPsrc case study

The optimisation procedure:

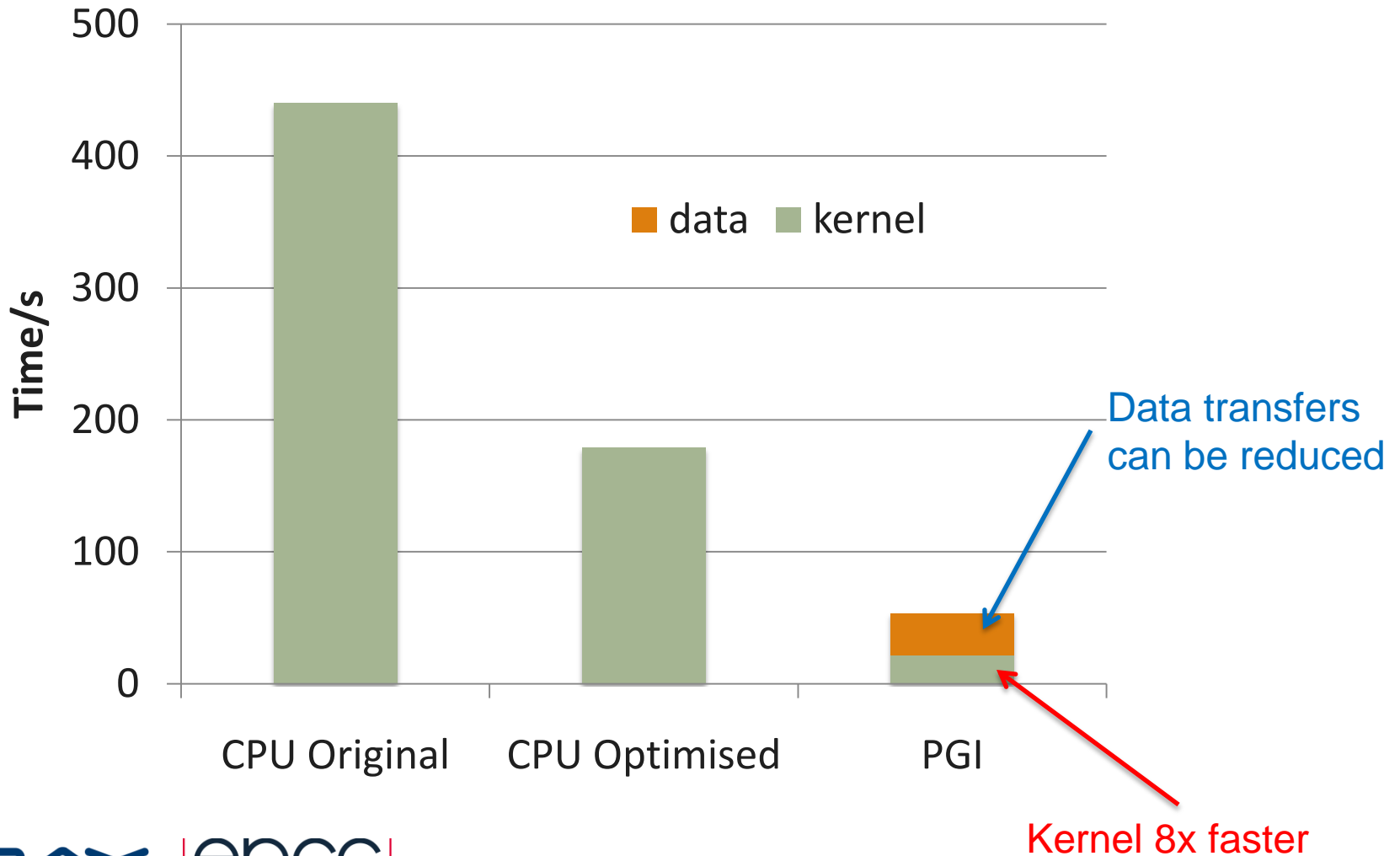
1. Optimise data movement
2. Refactor code to expose as much parallelism as possible
 - compiler feedback has to be read carefully
3. Make sure array accesses are coalescing
 - especially with single-level parallelism on outer loop
4. Privatise arrays explicitly for optimal index placement
 - trial and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

Ludwig description

- Simulates hydrodynamics of complex fluids
 - Lattice Boltzmann code, [Phys. Commun. 134 \(2001\) 273](#)
 - 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 but also with PGI accelerator and C
 - User must explicitly privatise to place extra index correctly for coalescing

Ludwig performance plot

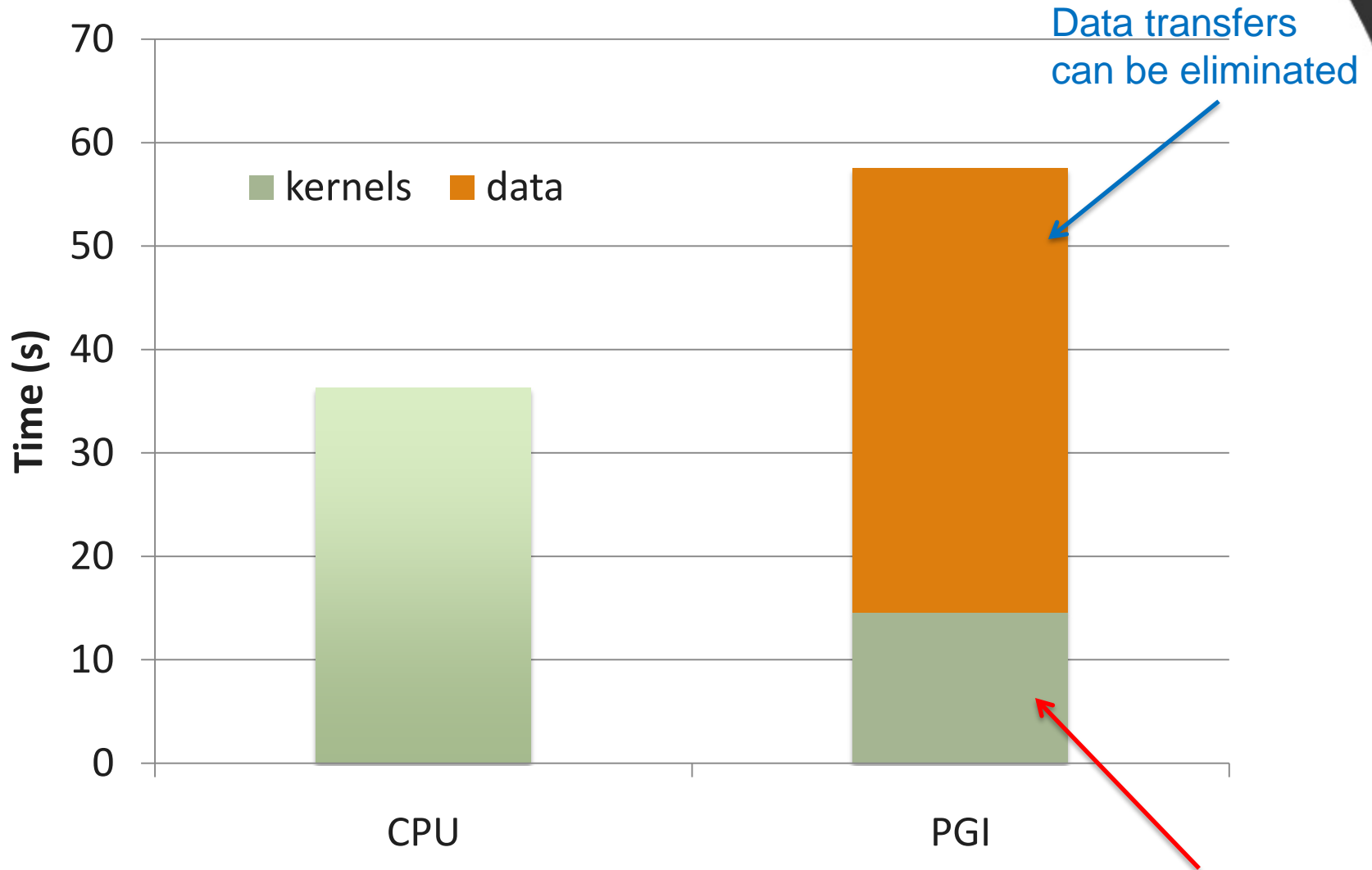
PGI Acceleration of Ludwig Collision Kernel 128³, 100 Iterations



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
 - Three main hotspot subroutines:
 - resid (50% of runtime), psinv (25%), rprj3 (9%)
 - Data arrays passed to/from subroutines at every iteration
- Accelerated with PGI and CCE OpenMP accelerator directives
 - Had to explicitly privatise with PGI (not CCE)
 - No performance tuning yet
 - PGI and CCE untuned versions perform comparably

NPB MG performance (class A, 256 x 256 x 256)



Summary

- Introduced accelerator directives: PGI and OpenMP in the CCE
 - Attractive programming model
 - Based on original Fortran, C and C++ codes
- Case study in accelerating the simple HPsrc kernel
 - 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
- Presented results for Ludwig and MG codes
 - Good speedups over CPU
 - Can significantly improve kernel performance with tuning

Upcoming performance boosts

- Accelerator directives programming model evolving rapidly
 - As is PGI and CCE compiler support
- Amongst the features expected in next few months:
 - Keeping data on GPU between subprogram calls
 - **mirror** (PGI), **acc_present** (OpenMP)
 - Very important when multiple kernels process same data
 - Data movements can currently wash out GPU performance boost
 - Accelerator-only data structures
 - **local**(PGI), **acc_res** (OpenMP)
 - e.g. when explicitly privatising arrays
 - CPU memory can be more limited than GPU
 - Asynchronous GPU kernels and data transfers
 - **async** (OpenMP)
 - Permits simultaneous use of CPU and GPU (multicore CPU not irrelevant)
 - Most codes will need algorithmic work to exploit this

Helping users exploit GPUs

- Users need good feedback and training
 - 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
 - Integrated performance analysis for GPU and CPU
 - 30 years experience of optimised vectorisation in the CCE
 - Optimised GPU libraries
 - Education: Cray GPU workshops (given by John Levesque)
 - [EPCC](#) (Edinburgh), Friday 22nd October
 - [HLRS](#) (Stuttgart), Monday 25th October

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
 - Luiz DeRose, John Levesque, James Beyer, David Oehmke...

And look out for our poster at SC10...

GPU Acceleration of Scientific Applications
Alan Gray¹ Alan Richardson^{1,3} Karthee Sivalingam¹ Alistair Hart^{2,4} Iain Bethune¹
¹EPCC, The University of Edinburgh ²School of Physics, The University of Edinburgh ³EAPS, Massachusetts Institute of Technology ⁴CRAY UK Limited

Until relatively recently, increases in the performance of CPUs (and in turn HPC systems) were largely achieved through increases in the clock frequency of the core. This trend has now reached it's limit mainly due to power requirements and heat dissipation restrictions. Today, cores are not getting any faster, but instead parallelism must be exploited within each chip for performance improvements. In commodity CPUs, the number of cores per chip is increasing, and the number of data elements which each core can operate on per clock cycle is also increasing.

Meanwhile, in recent years the computer gaming industry has driven development the Graphics Processing Unit (GPU), where parallelism is being exploited more aggressively: the silicon is largely dedicated to hundreds of simplistic cores, at the expense of controllers, caches, sophisticated etc. GPUs work in tandem with the CPU (communicating over PCIe) and are responsible for executing the

Chemics
The LUDWIG application uses Lattice-Boltzmann models to enable the simulation of the hydrodynamics of complex fluids in 3-D [2]. It permits the study of a wide range of systems, notably those incorporating "Liquid Crystal (Blue Phases)" which may be exploited in future flat-switching LCD devices. Other examples include systems under shear and fluid suspensions. The code is written in C, parallelised using MPI and scales well to many thousands of cores. The problem domain is a regular 3-D cell and the key target is the

fusion

ysics

The Standard Model of Particle Physics is known to be an extremely accurate theory, but does not account for all observed phenomena. Any cracks in the Standard Model, which point towards new physics, may be found by comparing experimental results with high-precision calculations performed using Lattice QCD.

To enable these comparisons, a matching factor must

Optimisations

Minimisation of data movement
The bandwidth from Global Memory can often be a bottleneck for applications. To achieve optimal bandwidth, the memory access pattern must satisfy the criteria necessary for accesses to be coalesced: typically this means that data must structured such that consecutive threads read consecutive memory addresses. Reorganising data structures to enable coalescing can have dramatic effects on performance.

Minimisation of expensive memory access patterns
Expensive Global Memory accesses can be reduced by utilising the on-chip memory spaces and registers, usage of which can be controlled within the CUDA code. It must be noted that increasing such usage may decrease the number of active threads per core thus degrading performance, so tuning must be done to find