

# Accelerator directives: a user's perspective

Alistair Hart, Harvey Richardson (*Cray Exascale Research Initiative, Edinburgh*),  
Alan Gray (*EPCC Exascale Technology Centre, Edinburgh*), Karthee Sivalingham (*EPCC*)

**ABSTRACT:** *Accelerated nodes will be increasingly important as we move towards exascale supercomputers. At present, a major barrier to the widespread exploitation of accelerators is the need to rewrite application kernels in specialist, low-level programming languages. These kernels are hard to maintain and require complicated performance tuning for each new problem size and accelerator architecture. Accelerator directives are an attractive alternative, allowing a single version of the source code to be compiled and run efficiently on a variety of current and future architectures (CPUs, GPUs and, looking forward, more general heterogeneous compute nodes). A directive-based approach is also a more familiar model to many programmers and should yield much greater productivity for the application developer than a specialist, low-level language. In this talk we discuss our experiences using two sets of directives: those from PGI, and the proposed OpenMP extension being implemented in the Cray Compilation Environment. We look at what the directives can do and how they can be used in practice. For some simple examples, we find the performance to be comparable to CUDA.*

## Introduction

Exascale architectures will necessarily contain large numbers of multi-socket, multi-core nodes. To increase node performance, accelerators will be an important component. In future we expect that GPU functionality will be delivered by architectures with full-featured cores combined with many simpler GPU-like cores on the same die. We need a flexible programming paradigm that can embrace this evolving architecture, especially as we move away from the current distinct host (CPU) and device (GPU) memory spaces, connected by a relatively slow (high latency, low bandwidth) PCIe bus.

Until recently, GPU programming required the developer to use low-level languages like CUDA. Optimising these kernels for a given GPU required significant recoding and a good understanding of the underlying architecture. This reduced the codes' portability and made them very difficult to maintain.

A new, directive-based approach is now emerging that promises to overcome a lot of these problems. The Cray Exascale Research Initiative and EPCC Exascale Technology Centre have a joint Edinburgh-based project exploring this programming model as part of their wider investigation into how to scale applications onto exascale supercomputers. Here we report some of our initial experiences accelerating codes using the PGI and OpenMP accelerator directives.

## Accelerator directives

Accelerator directives are specially formatted, non-executable statements (comments in Fortran, pragmas in C and C++) inserted in the original source code. Using these directives, we can mark out kernels of the code (usually loop nests) that are suitable for execution on the accelerator. Other directives allow us to guide the compiler in tuning the kernel performance (e.g. optimising data movement, loop scheduling and cache usage). Non-accelerating compilers ignore these directives. There are several sets of accelerator directives available. Here we concentrate on two: PGI, being perhaps the most widespread at present, and the draft OpenMP standard that is currently being implemented in the Cray Compilation Environment (CCE).

The PGI and OpenMP directives have many similarities. PGI accelerator directives begin with `!$acc` in Fortran or `#pragma acc` in C<sup>1</sup>. Whilst specific to the PGI accelerating compilers, these directives have proved powerful and also helped provide a template for the OpenMP accelerator directives. The OpenMP directives start with `!$omp acc` in Fortran or `#pragma omp acc` in C and C++. Complete support for the draft standard is expected in the CCE by the end of 2010. As the subcommittee developing these directives contains representation from all the major vendors, including Cray (co-chair), PGI and CAPS, we expect other vendors to support the OpenMP standard in due course. CCE support will also track the standard as it develops.

In the following, we discuss three examples where we used directives to accelerate scientific codes. Our methodology is similar in all cases. We first profile the code on the host. Kernels suitable for acceleration should take a significant portion of the runtime and have a large amount of loop-based parallelism. There loops should avoid complicated branching and there should preferably be more than one level of loops. High level loop iterations should be independent, to map to different GPU thread blocks. Determining if this is true is often difficult but is a problem understood by developers already using OpenMP to take advantage of multiple threads on a host. Cray is also developing tools to help the user with this. Low-level loops should be vectorisable, to map onto the SIMD-like thread blocks. The CCE has a long history of identifying opportunities for vectorisation.

Once directives are added, compiler feedback (via `-Minfo=accel` for PGI) is used to gain an understanding of data movement to the GPU. We can then add directives to optimise data movement (`-ta:time` gives runtime information on this) and then to tune performance.

---

<sup>1</sup> <http://www.pgroup.com/resources/accel.htm>

## HPsrc

The HPsrc code was developed for lattice QCD<sup>2</sup> and uses perturbative calculations (integrals of double-precision complex-valued functions) to tune parameters used in large-scale Monte Carlo simulations. The code is written in Fortran90.

The majority of the runtime (around 80%) is spent in calculating weighted sums of complex exponentials. A simplified version of the kernel is shown in Fig. 1. Typically each sum contains `nterms=8000` terms (used here), and we must calculate the sum separately for `npoints` independent data points, where `npoints` is large, but can be split into smaller chunks if required.

PGI directives were used to accelerate the original source code. Loop dependencies led to the compiler selecting a single-level parallelism on just the `p`-loop. An equivalent CUDA kernel (called via a C wrapper) was written. CUDA does not support complex data types, so separate arrays were used for real and imaginary parts. A similar rewrite was needed in the host code to work around limitations of accelerator intrinsic support of complex data types in our version of pgfortran (10.8.0), now being addressed by PGI. To maximise performance, array `k` was reordered so that `p` was the fastest index, ensuring coalescence of GPU memory accesses. This improved performance by a factor of 2 to 3. Array `p3` was also explicitly privatised with `p` the fastest index, reducing runtimes by a further factor of around 2.

The speeds of the CPU (compiled with `-O3`) and directive-derived GPU kernels are shown in Fig. 2, relative to the CUDA version. Asymptotically, the PGI performance is around 80% of CUDA on an NVIDIA Fermi C2050 card. The PGI accelerated version is around 33 times faster than that on the single-core CPU. Data transfer times were negligible in this example. The PGI directive performance was improved further by explicitly privatising array `p3` as above and splitting the `p`-loop into two sections (the latter including just the sum). The compiler then used two-level parallelism in the first section and the overall performance was now marginally *better* than the best single-level-parallelised CUDA kernel.

We note that we started this project using a Tesla M1060 card and spent a lot of time hand-tuning the CUDA kernel (especially the shared memory usage). On the Fermi card, this had no significant effect, presumably due to automatic usage of the new cache. The simplified kernel has also been successfully accelerated using OpenMP accelerator directives and the CCE. No workarounds were needed as the CCE accelerating compiler supports complex arithmetic.

## NPB Multigrid benchmark

The NAS NPB parallel benchmarks are used to test a variety of parallel programming paradigms<sup>3</sup>. The MG multigrid benchmark is particularly suited to GPU acceleration, being relatively simple and having a profile dominated by a few computationally-intensive subroutines. In contrast to the HPsrc kernel, the data structures (a set of 3-D arrays) are explicitly passed between subroutines in the Fortran code.

We accelerated the code with PGI directives, using the OpenMP version as a template. In some cases, temporary arrays in loop nests had to be explicitly privatised using additional array indices to ensure correctness.

In Fig. 3 we compare performance of the code on the CPU (single thread) with the PGI-accelerated version on a Fermi C2050 card. At present, there is no mechanism for maintaining data on the device between subroutine calls, so every accelerated region required significant data movement in and out. We used the `-ta=time` compiler option to separately time GPU initialisation, data

```
INTEGER, PARAMETER :: Nterms = 8000
COMPLEX(kind=dp) :: k(0:3,Order,Npoints)
COMPLEX(kind=dp) :: f(Nterms)
INTEGER :: yxv(0:3,Order,Nterms)

!$acc region
!$acc do private(p3)
DO p = 1,Npoints
  DO i = 1,Nterms
    phase3 = CMPLX(0,0.5d0) * &
              SUM(k(:, :, p)*yxv(:, :, i))
    p3(i) = f(i) * EXP(phase3)
  ENDDO
  vtx(p) = SUM(p3)
ENDDO
```

Figure 1. The simplified HPsrc kernel

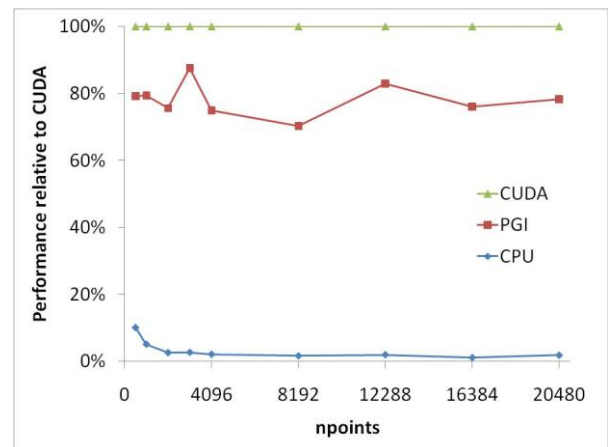


Figure 2. Relative performance of PGI accelerator directives and host code for the simplified HPsrc kernel with increasing problem size.

two-level parallelism in the first section and the overall performance was now marginally *better* than the best single-level-parallelised CUDA kernel.

We note that we started this project using a Tesla M1060 card and spent a lot of time hand-tuning the CUDA kernel (especially the shared memory usage). On the Fermi card, this had no significant effect, presumably due to automatic usage of the new cache. The simplified kernel has also been successfully accelerated using OpenMP accelerator directives and the CCE. No workarounds were needed as the CCE accelerating compiler supports complex arithmetic.

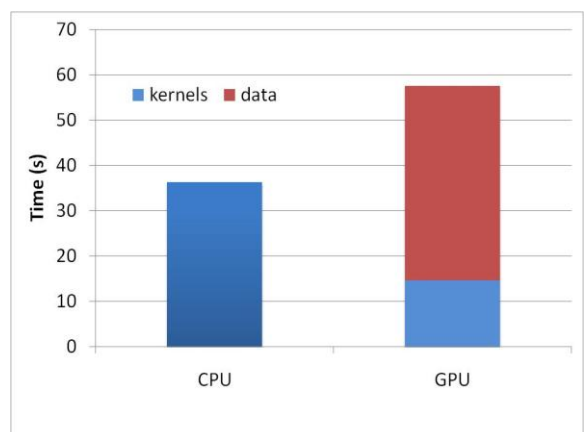


Figure 3. Performance for the MG benchmark relative to a single thread on the host for the CLASS=B problem size.

<sup>2</sup> A. Hart, G.M. von Hippel, R.R. Horgan, E.H. Müller, Comput. Phys. Commun. 180 (2009) 2698 [doi:10.1016/j.cpc.2009.04.021].

<sup>3</sup> Version 3.3.1, <http://www.nas.nasa.gov/Resources/Software/npb.html>

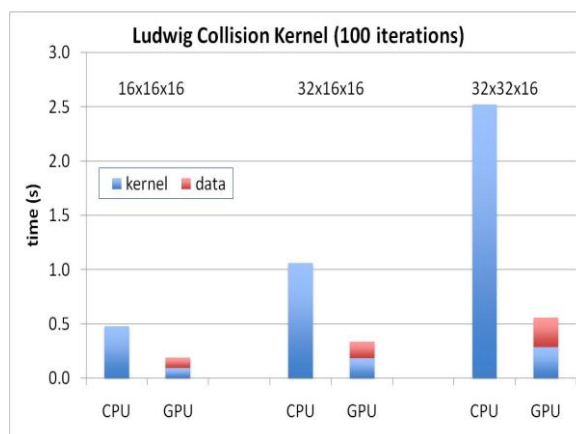
movement and accelerator kernels. Subtracting data transfer and initialisation, the PGI-accelerated version was around 2.5 times faster than on the host. We are currently investigating how PGI directives for caching and loop scheduling can improve this. We also have a working version of the benchmark parallelised using OpenMP accelerator directives. The CCE did not require explicit privatisation of temporary arrays.

### Ludwig

Ludwig uses Lattice-Boltzmann models to simulate the hydrodynamics of complex fluids<sup>4</sup>. The code is written in C and MPI and scales well to many thousands of cores. The problem domain is a regular 3-D grid with the collision kernel taking around 90% of the runtime.

This kernel was accelerated using PGI directives, refactoring as necessary. In some cases, e.g. temporary arrays in loop nests, this also improved the CPU performance by 50% with the PGI compiler (but not with the CCE). The PGI compiler doesn't yet support subprogram calls within accelerated regions, so these were manually inlined. Non-trivial use of structs, e.g. arrays within structs or nested structs, was also problematic and data was copied to standard C arrays. Loop dependencies were complicated and the `independent` clause was needed to guide acceleration. As for HPsrc, large performance gains were made by explicitly privatising some temporary arrays rather than by using the `private` clause. This reflected the loop structure of the code, where parallelisation was only possible on the outer loop.

Fig. 4 compares performance for the PGI accelerated version to that of a single CPU core for various local grid sizes. The GPU results have been decomposed into data transfer time and compute time. For the larger problem sizes, we see this kernel is accelerated by a factor of around 5 relative to the (refactored) host code.



**Figure 4. Performance of the accelerated Ludwig collision kernel relative to a single thread on the host, for three different problem sizes.**

### Conclusions

Compiler directives present a very attractive way to exploit accelerators, enhancing the original source code using a familiar, directive-based programming model. In particular, it gives the programming environment much more scope to produce optimised kernels for a range of different target architectures from the same source code. Cray is taking a lead in these developments, enhancing the Cray Compilation Environment to support the draft OpenMP accelerator directives and developing a suite of scoping and optimisation tools to take full advantage of parallelism in applications.

We presented three representative examples of codes (Fortran and C) that we have accelerated using PGI and OpenMP directives. In all three cases, we see that accelerator directives can produce efficient kernels, sometimes matching the performance of CUDA, and with a lot less development effort. As expected for a maturing programming model, the performance of the directive-accelerated kernels is better for structurally simpler kernels (e.g. HPsrc), but our results do not include performance-tuning directives which we are now investigating. The performance of kernels will also rapidly improve as accelerating compilers mature.

It is clear from our work (MG and Ludwig) that the ability to retain data on the device between subprogram calls will be key to accelerating many applications. Both PGI and OpenMP developers recognise this and support is imminent from both PGI and Cray. Automatic inlining with the compiler would also help here.

When optimising, the first step should be to expose as much parallelism as possible, including via explicit privatisation of arrays. Arrays should also be reordered to ensure coalescence, based on the compiler loop scheduling feedback. In some cases the PGI compiler already gives guidance on this. We also encountered directive interoperability issues when we tried to compile codes that contained `omp`, `acc` and `omp acc` directives. To avoid extensive use of a preprocessor, additional compiler options would be needed, for instance to recognise `omp` directives but ignore `omp acc`.

Impressive though they are, our performance figures underline that a multicore CPU often performs comparably to a GPU. The `async` directive clause will allow users to overlap computation and/or communication on the GPU and CPU. Exploiting this efficiently will, however, be a challenge in many codes, requiring algorithmic changes.

Accelerator directives should become a popular way of programming accelerators and make GPU technology more attractive to application developers. The main performance advantage is that the compiler has much more scope to produce optimised kernels from the source code. This is not to be undervalued: we have already seen how work optimising CUDA code for Tesla was rendered unnecessary by improvements in the Fermi design.

The authors would like to thank colleagues within Cray and EPCC for valuable discussions and advice. We are also grateful to Mathew Colgrove (PGI) and the PGI forum for very good feedback on a wide range of issues.

<sup>4</sup> J.-C. Desplat, I. Pagonabarraga, P. Bladon, *Comput. Phys. Commun.* 134 (2001) 273 [[doi:10.1016/S0010-4655\(00\)00205-8](https://doi.org/10.1016/S0010-4655(00)00205-8)].