The Productivity, Portability and Performance of OpenMP 4.5 for Scientific Applications Targeting Intel CPUs, IBM CPUs, and NVIDIA GPUs

Matt Martineau(✉) and Simon McIntosh-Smith

HPC Group, University of Bristol, Merchant Venturers Building,
Woodland Road, Bristol BS81UB, UK
{m.martineau,cssnmis}@bristol.ac.uk

Abstract. This research considers the productivity, portability, and performance offered by the OpenMP parallel programming model, from the perspective of scientific applications. We discuss important considerations for scientific application developers tackling large software projects with OpenMP, including straightforward code mechanisms to improve productivity and portability. Performance results are presented across multiple modern HPC devices, including Intel Xeon, and Xeon Phi CPUs, POWER8 CPUs, and NVIDIA GPUs. The results are collected for three exemplar applications: hydrodynamics, heat conduction and neutral particle transport, using modern compilers with OpenMP support. The results show that while current OpenMP implementations are able to achieve good performance on the breadth of modern hardware for memory bandwidth bound applications, our memory latency bound application performs less consistently.

Keywords: OpenMP-4 · High-performance-computing · Mini-apps

1 Introduction

The diversification of modern architecture has lead to increasing demand for parallel programming models that improve the productivity and future portability of large scientific applications. An implicit expectation is that parallel programming models will provide the features that are necessary to achieve near optimal performance, with some understanding that there is a trade-off between improved productivity and portability, and absolute performance.

OpenMP provides an extensive feature set that allows application developers to tune their applications for performance, while providing an intuitive interface that enables relatively un-intrusive parallelisation of applications. The performance achieved in practice is dependent not only on the features provided by the specification, and the developer's use of those features, but by the implementation provided by the compiler vendors.

© Springer International Publishing AG 2017
B.R. de Supinski et al. (Eds.): IWOMP 2017, LNCS 10468, pp. 185–200, 2017.
DOI: 10.1007/978-3-319-65578-9_13

It is essential that both computer scientists and domain scientists are able to effectively explore the potential programming environments without the constraints of porting large scientific applications. Mini-apps are widely accepted as a powerful tool for evaluating the performance of parallel programming models, but it is essential that a broad range of performance profiles are assessed to observe the edge cases of performance exposed by production applications [1].

In this research we will be using a suite of mini-apps that fall under the `arch` project[1], developed at the University of Bristol. Each of the mini-apps has been developed to provide research tools for computer scientists to support applications scientists in porting real codes. Although an understanding of the core Physics of each package is not required in this paper, we will discuss the performance profile of each application as we present results.

2 Contributions

It is expected that the results of this research will be of use to domain scientists and application architects looking to determine if OpenMP is a good fit for their project, and may offer support to developers already using OpenMP for their software projects. Through evaluating a range of proxy applications on cutting edge hardware, we provide insights into the differences between available OpenMP implementations that can feed into future optimisation.

This research specifically contributes the following:

- Specific code suggestions for improving productivity and portability of large scientific applications, based on real porting experience.
- A discussion of limitations of the specification, and important differences between OpenMP implementations.
- An extensive performance analysis of OpenMP 4 ports of three distinct application classes: explicit hydrodynamics, sparse linear algebra, and Monte Carlo neutral particle transport, on modern HPC hardware: Intel Xeon and Xeon Phi CPUs, IBM POWER8 CPUs, and NVIDIA K20X and P100 GPUs.

3 Productivity

The authors have encountered issues with productivity that we expect will be encountered by developers using OpenMP 4 for non-trivial applications.

3.1 Structured and Unstructured Data Regions

Maintaining data resident on a device is generally one of the most important considerations for offloading to accelerators. We previously discussed the difficulties that are encountered when attempting to copy data to and from the device using the structured `target enter data` directive [2].

[1] https://github.com/uob-hpc/arch.

Listing 1.1. OpenMP 4.0 approach to copying data for an application.

```
double* density0 = (double*)malloc(sizeof(double)*nx);

#pragma omp target enter data map(to: density0[:nx])
{
  for(int tt = 0; tt < ntimesteps; ++tt) {
    // Do work
  }
}
```

With OpenMP 4.0, the initial copying of resident data into the device data environment would be approached as shown in Listing 1.1. When the number of arrays increases, this approach becomes less readable and maintainable. As the structured data regions only operate upon a structured block, the application structure will be limited if developers want to avoid redundant data movement.

Listing 1.2. OpenMP 4.5 approach to copying data for an application.

```
void allocate_data(double** array, size_t len) {
  (*array) = (double*)malloc(sizeof(double)*len);

  double* local_arr = *array;
  #pragma omp target enter data map(to: local_arr[:len])
}

allocate_data(&density, nx);

for(int tt = 0; tt < ntimesteps; ++tt) {
  // Do work
}
```

The unstructured data mapping introduced in OpenMP 4.5 allowed us to combine the allocation and mapping into a method, as seen in Listing 1.2. This significantly reduced the code duplication, and improved the productivity of our porting efforts by abstracting OpenMP data allocations from the core of the codes. Another benefit is that when arrays were resized during development it was only necessary to propagate the change to a single location.

3.2 Copying Members of Structures

The OpenMP specification does not handle the copying of pointer members of a structure into the device data environment. In many codes, pointer data is exclusively passed around in structures, and developers generally want to be able to access that data in the manner demonstrated in Listing 1.3. Unfortunately, the specification does not state how the pointer members of the structure should be copied onto the device. The Cray compiler implementation of OpenMP 4 currently emits a compile-time error, whereas the Clang compiler supports the form of Listing 1.3, in spite of the limitation in the specification.

Listing 1.3. Mapping an array section that is a member of a structure.

```
#pragma omp target teams distribute parallel for \
    map(some_struct.a[:len])
for(int ii = 0; ii < n; ++ii) {
  some_struct.a[ii] = 0.0;
}
```

The consequence of this missing functionality is that codes currently attempting to achieve portability between compilers with OpenMP 4.5 will have to deserialise the pointer members of structures before they are mapped, and change all kernel accesses to reference the private variables, as seen in Listing 1.4. This significantly limits productivity for large applications, especially where Structure of Arrays style data structures have been adopted.

Listing 1.4. Privatising an array section that is a member of a structure and then mapping it.

```
double* a = some_struct.a;
#pragma omp target teams distribute parallel for map(a[:len])
for(int ii = 0; ii < n; ++ii) {
  a[ii] = 0.0;
}
```

All of the applications we have ported, including the mini-apps investigated in this research, pass their pointer variables through the kernel interfaces, rather than copying them into private variables before each kernel invocation. This approach still requires a significant refactoring when porting codes, but minimises the resulting overhead in terms of new lines of code.

3.3 Tools

Access to high quality tooling is one of the most significant influences on productivity. While porting the suite of mini-apps presented in this research, the process was supported by the compiler vendors' tool suite. For the CPU, tools such as `VTune` and `CrayPat` are all compatible with OpenMP, and provide detailed OpenMP-specific insights. The NVIDIA CUDA toolkit, which includes `nvprof`, also works with the OpenMP 4 implementations discussed in this paper. Application developers can expect this tool support to improve even further with future releases of the OpenMP specification as a new tools interface is set to be included in version 5.0 of the specification [3].

4 Portability

OpenMP 4.5 is becoming increasingly accepted within the community, and the implementations that can target heterogeneous architectures are constantly improving. Intel, Cray Inc., IBM, and GNU, are all actively developing OpenMP

support for the newest features of OpenMP. The thread parallelism features of OpenMP 3.0 are mature and well supported, whereas the offloading features were added more recently, and introduced new challenges to implement in a compiler.

4.1 OpenMP Compilers

There are many OpenMP compilers available, and we discuss and evaluate a cross section of the most popular.

The Intel Compilers provided the first vendor-supported OpenMP 4 implementation, targeting the Intel Xeon Phi Knights Corner architecture, but Intel has since moved away from the offloading models for their future architectures. In spite of this, Intel's OpenMP 4.5 compliant compiler (version 17.0+) can be used to target the Intel Xeon and Intel Xeon Phi processors.

The Cray Compilers provided the first vendor-supported implementation of OpenMP 4 that allowed developers to target NVIDIA GPUs. Cray subsequently ceased development of their OpenACC implementation, suggesting that they see OpenMP as the future parallel programming model for targeting their heterogeneous supercomputers. The Cray compiler (version 8.5.7) is not yet OpenMP 4.5 compliant, although it is fully OpenMP 4.0 compliant and supports a number of OpenMP 4.5 features.

The Clang/LLVM Compiler Infrastructure was recently forked to develop OpenMP 4.5 support for targeting NVIDIA GPUs by IBM Research. The fork of the compiler[2] is now OpenMP 4.5 compliant, and the support is being actively patched into the main trunk of the Clang front-end [4]. Although the implementation was developed from the perspective of running on the POWER8/POWER9 and NVIDIA GPU nodes, such as those being installed in Sierra and Summit [5], the compiler will also allow users to compile for NVIDIA GPUs hosted on X86 platforms. One limitation for scientific application developers is that Clang is a C/C++ front-end for LLVM. A team at the Portland Group are currently implementing an open source Fortran front-end, codenamed Flang, which will eventually support OpenMP 4.5.

The PGI Compilers do not yet support any OpenMP 4.0 features, but provide full support for OpenMP 3.0. The compilers support an alternative to OpenMP, OpenACC, which is similar except for some additional features and limitations, but allows users to offload to both CPUs and GPUs.

The XL Compilers are a closed-source compiler suite developed by IBM, and deployed with the POWER architecture, that will support OpenMP 4.5 in time for the installation of the Summit and Sierra supercomputers. The Clang effort for targeting NVIDIA GPUs is more advanced at this stage, and the research is being fed directly into XL. A subset of OpenMP 4.5 features are supported in the version 13.1.5, which we had access to, however support was not available for the `reduction` clause on `target` regions, or `atomic` operations, making it impossible to collect results for XL targeting NVIDIA GPUs in this research.

[2] https://github.com/clang-ykt.

The GNU Compiler Collection has officially supported OpenMP 4 offloading since version 5.0, but feature-rich implementations that target specific architecture such as GPUs are still not available. Offloading support exists for AMD GPUs via HSA, but the support is limited to a single combined construct with no clauses. The compiler is capable of targeting Intel Xeon Phi KNLs with OpenMP 4.5, and GNU are currently working on an OpenMP 4.5 implementation that can target NVIDIA GPUs, as mentioned in the *in progress* documentation for GCC 7.1.

4.2 Homogeneous Directives

We have previously shown that it is not yet possible to write a single homogeneous line of directives to achieve performance portability with OpenMP [2,6]. Standardisation of the compiler implementations is important for future performance portability, for instance, the newest Clang implementation automatically chooses optimal team and thread counts, so that the developer does not have to list architecture-specific values. This is one of many issues with standardisation:

- The impact of the `simd` directive will vary significantly between architectures. For instance, on CPUs it will generally command the compiler to generate SIMD instructions, whereas on the GPU it might tell the compiler to spread the iterations of a loop across the threads in a team.
- Setting a `thread_limit` and `num_teams` for one architecture means you cannot choose the default compiler behaviour for other architectures.
- As seen with the porting exercises, there can be significant performance implications when using the `collapse` statement on different architecture.

Achieving performance portability with a single codebase requires the preprocessor, or abstractions above OpenMP. We are hopeful that future versions of the specification will introduce conditional capabilities to make it possible for developers to write a homogeneous set of directives at the loop-level.

4.3 OpenMP 4.0 to OpenMP 4.5

The authors of this paper strongly believe that OpenMP implementations need to support some key features of version 4.5 to avoid future portability issues. Having ported scientific codes to use OpenMP 4.0 and OpenMP 4.5, we have come across some compatibility issues between the versions. Developers who are using compilers that target OpenMP 4.0 compliant implementations should be aware that these pitfalls can lead to subtle bugs caused by implicit behaviour.

In OpenMP 4.0, the default copying behaviour of scalar variables was to copy them to and from the device, when entering and leaving a `target` region. This implicit behaviour was as if the `map(tofrom:`*`scalar-variable`*`)` clause had been included on the target region. In OpenMP 4.5 the default behaviour is that variables are declared `firstprivate`, and so the scalar variable will not be copied back from the device. Developers who have written their applications to rely upon the scalar variables being returned at the end of the `target` region will encounter potentially difficult to diagnose application bugs.

5 Mini-app Studies

No algorithmic changes are present between versions of the mini-apps, which ensures that they resolve to within tolerance of a single result having executed the same computational workload, regardless of the OpenMP implementation or target device. The purpose of this section is not to compare the different architectures or the algorithms, for which discussions can be found in other literature [2]. This section is instead intended to consider the differences in performance achieved by the different OpenMP compilers targeting the same architecture. Developers familiar with OpenMP may expect there to be minor variations between compiler implementations, but we aim to expose some cases where more significant variance can be observed.

5.1 HPC Devices

The performance evaluation in this paper considers five modern HPC devices, which feature, or will feature, in some of the largest supercomputers in the world.

Where possible, we compare OpenMP to optimised MPI and CUDA ports of the mini-apps, allowing an objective assessment of the performance of the OpenMP implementations compared to the best performance achievable. Subsequent performance evaluation is conducted with the compilers in Table 1.

5.2 Hot and Flow

The `flow` mini-app[3] is a 2d structured Lagrangian-Eulerian hydrodynamics code, that explicitly solves the Euler equations using a chain of threaded kernels executed across the computational mesh. The application contains little branching, and a minor load imbalance with the artificial viscous terms, but this does

Table 1. The HPC devices used in this performance evaluation, where Intel Xeon Broadwell means dual socketed 22 core CPUs, POWER8 means dual socketed 10 core CPUs, and *Mem BW* is the maximum benchmarked memory bandwidth [7]. The clang-ykt compiler was built with all commits up to date 30th May 2017.

Device name	Mem BW	Compiler
Intel Xeon Broadwell E5-2699 v4	120 GB/s	ICC 17.0.2, GCC 6.1.0, PGI 17.3.0, CCE 8.5.7
NVIDIA K20X	180 GB/s	CUDA 8.0 + GCC 4.9.3, CCE 8.5.7, clang-ykt
IBM POWER8	298 GB/s	XL 13.1.5, GCC 6.1.0, PGI 17.3.0
Intel Xeon Phi Knights Landing 7210	440 GB/s	ICC 17.0.2, GCC 6.1.0
NVIDIA P100	500 GB/s	CUDA 8.0 + GCC 4.9.3, CCE 8.5.7, clang-ykt

[3] https://github.com/uob-hpc/flow.

not generally affect performance. Due to the low computational intensity and regular mesh access, `flow` is memory bandwidth bound.

The `hot` mini-app[4] is a 2d heat diffusion code, that uses the Conjugate Gradient method to implicitly solve the sparse linear system. The application is memory bandwidth bound, and comprised of short linear algebra kernels, including a sparse matrix-vector multiplication and several reductions. The kernels are highly data parallel, with low register usage and no branching.

Both applications are optimised to achieve roughly 70–80% of achievable memory bandwidth in the best case on the target architecture.

Porting: Both applications are comprised of multiple simple kernels, and parallelisation of those kernels was achieved with `#pragma omp parallel for`, or `#pragma omp target teams distribute parallel for` for offloading. Data allocations are handled by the `arch` project's data allocation wrappers, so a simple overload of the wrappers for OpenMP 4 meant data could be mapped into the device data environment at allocation, as described in Sect. 3.1. The `reduction` clause was required in both applications, and, due to the specification implicitly mapping the scalar reduction variable as `firstprivate`, it was necessary to explicitly `map` the reduction variable as `tofrom` to copy the results back from the device. Also, vectorisation was forced on the CPUs and KNL using the `#pragma omp simd` clause on the inner loops.

A major limitation with the current specification is that it was not yet possible to express CPU and GPU parallelism at the same time for our mini-apps, meaning that multiple instances of the directives were required, as discussed in Sect. 4.2. For the GPU ports, we noted that using `collapse` on the outer loops of the applications' kernels resulted in significantly reduced performance when compiling with CCE, for instance `hot`'s runtime worsened from 44 s to 49 s on the P100. This is due to the way that CCE maps the iteration space to the GPU's threads, but serves as a case where the `collapse` clause can have unexpected negative influences on performance. The effort to port both applications was minimal, and amounted to roughly two additional lines of code per kernel.

Problem Specification: The mesh size for both applications is 4000^2, a large but realistic mesh size, and each application starts with a timestep of 1.0×10^{-2}s for the two test cases. The `hot` test case sets up a high density, high temperature region next to a low density, low temperature region. The `flow` test case sets up a two-dimensional interpretation of the sod shock problem, where an immobile square of high density, high energy fluid is surrounded by low density material.

Intel CPU and KNL Results: The Intel Xeon Broadwell (BW) results in Figs. 1 and 2 were highly consistent between compilers. Intel and flat MPI performed the best, and the largest performance difference was seen with GCC, which required 1.03x the runtime compared to the Intel compiler.

[4] https://github.com/uob-hpc/hot.

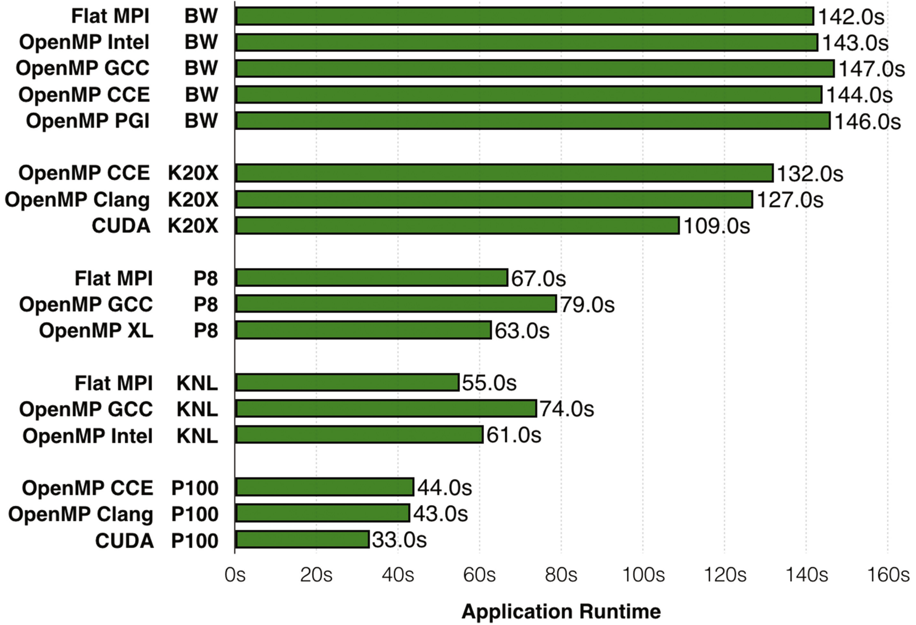

Fig. 1. The results of running `hot` on the target HPC devices. Devices are ordered from least to greatest achievable memory bandwidth.

On the KNL, application data was placed in MCDRAM, improving the runtime by roughly 5.0x compared to DRAM, as both applications are memory bandwidth bound. The MPI results are shown for running 128 ranks, whereas the OpenMP implementations performed better when using all four hyperthreads for 256 threads total. OpenMP `hot` experienced a 1.11x performance penalty compared to flat MPI, due to improved decomposition of the problem into cache seen with the MPI implementation. For `flow`, the difference was not significant.

OpenMP `hot` compiled with GCC suffered a 1.21x performance penalty compared to Intel, while OpenMP `flow` compiled with GCC suffered a penalty of 4.28x. Disabling vectorisation with the Intel compiler resulted in a runtime equivalent to GCC, suggesting that a lack of vectorisation accounts for the performance difference, in spite of the use of the `simd` directive.

POWER8 Results: On the POWER8 CPU we found 160 threads, or 8 Simultaneous Multi-Threads (SMT) per core, was optimal, and OpenMP compiled with XL was fastest for `hot`, while flat MPI was fastest for `flow`. GCC experiences a significant performance penalty of roughly 1.25x compared to XL for both `hot` and `flow`, which is significantly more than seen with the Intel CPU.

NVIDIA GPU Results: The CUDA results included for the NVIDIA GPUs represent an upper bound on performance for each mini-app. On the K20X, CCE

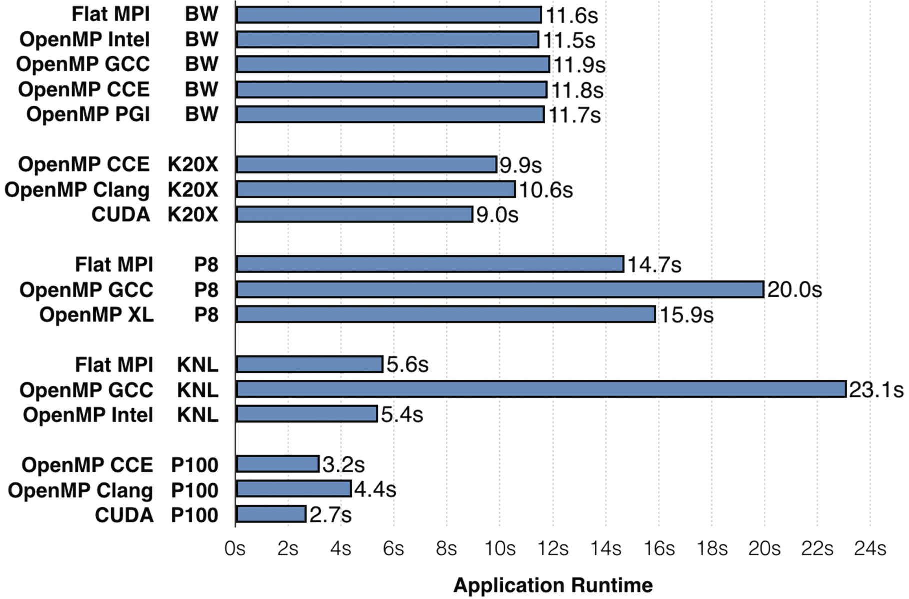

Fig. 2. The results of running `flow` on the target HPC devices. Devices are ordered from least to greatest achievable memory bandwidth.

achieved impressive performance for `flow`, requiring only 1.10x the runtime of CUDA, while `hot` was slightly less efficient at 1.20x compared to CUDA. When compiled with Clang, `flow` and `hot` both required 1.17x the runtime of CUDA. The performance penalty for OpenMP compared to CUDA was at worst 1.20x, which is an impressive result and would be tolerable for the improvements to portability and productivity offered by the programming model.

We observed a larger performance difference on the newer P100 GPUs, with the worst case being `flow` at 1.63x, but we feel that this is likely a performance bug given the results with other combinations, and continue to investigate the root cause. Apart from this result, the performance difference increased to around 1.25x to 1.30x, a significantly higher variation than seen on the CPU.

5.3 Neutral

The `neutral` mini-app[5] is a new Monte Carlo Neutral Particle Transport application that tracks particle histories across a 2d structured mesh [8,9]. The application has high register utilisation, and inherently suffers from load imbalance at the intra and inter node level. The algorithm parallelises over the list of particle histories, each of which is in principal independent. Particle histories exhibit a dependency on the computational mesh, to store tallies of the energy deposited

[5] https://github.com/uob-hpc/neutral.

throughout the space, which means the application suffers from random memory access and sensitivity to memory latency. At this stage in the mini-app's development, there is not an optimal MPI implementation, and so results for MPI are excluded.

Porting: Given that there is a single computational kernel, the porting process was straightforward and fast for all target architectures, following the same approach as for `flow` and `hot`. The only challenge when porting the application was that it depends upon a library, Random123 [10], for random number generation (RNG), which meant it was necessary to persuade the implementations to compile that code correctly. A load imbalance between threads is caused by the varying amounts of work for each history, and so we used `schedule(guided)` to optimise this, generally achieving a 5–10% improvement.

When targeting NVIDIA GPUs, adding `simd` to the combined construct, as `#pragma omp target teams distribute parallel for simd`, was essential to achieve good performance with CCE, improving the result from 211 s to 11 s on the P100. The reason that this directive is required is that CCE relies upon the kernel being vectorisable, and this particular kernel was so complex it needed the `simd` directive as a guarantee that there were no dependencies.

Problem Specification: The test case chosen for the `neutral` mini-app is the *center square problem*, where there is a region of high density material in the center of a low density space. A square neutron source is placed in the bottom left of the space, with all particles having a starting energy of 1 MeV, considering particle histories for 10 timesteps of length 1.0×10^{-7}s.

Intel CPU and KNL Results: The results shown in Fig. 3 are significantly less consistent between the compiler vendors than seen with either `flow` or `hot`. CCE required 1.18x the runtime of Intel and GCC required 1.98x the runtime, on the Intel Xeon Broadwell, which is significantly less optimal than we would have expected. The PGI compiler achieved worse than serial performance and, through the use of the *Minfo* flag, we determined it was transforming the `atomic` operations into `critical` regions. The application invokes billions of `atomic` operations, and so this serialisation is highly inefficient. We tested the issue further by removing the `atomic` operation, and the PGI result improved.

On the KNL we observed a 1.43x performance penalty for using GCC compared to the Intel compilers. We know that vectorisation is not the cause in this instance, and hypothesise that this is due to the way that registers are allocated by the compilers, which we know the application is sensitive too.

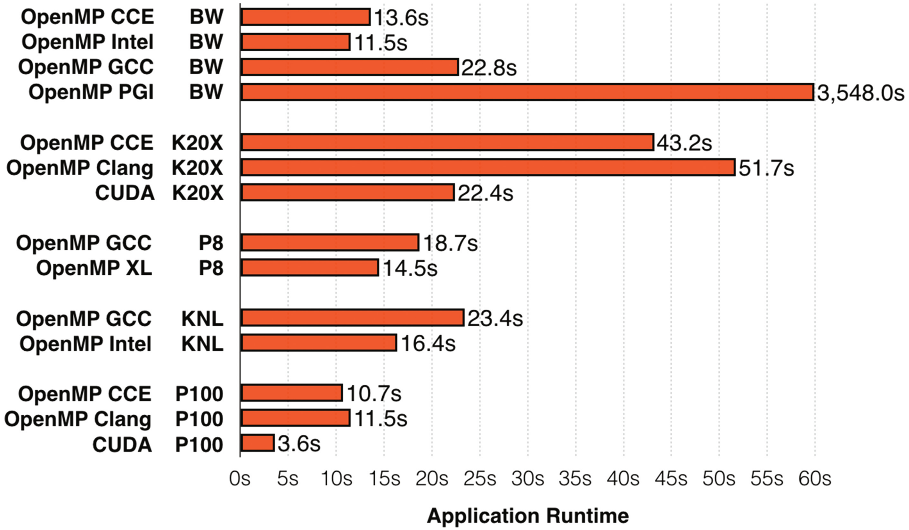

Fig. 3. The results of running `neutral` on the target HPC devices. Devices are ordered from least to greatest achievable memory bandwidth.

POWER8 Results: As with the other mini-apps, we observed a significant performance penalty for compiling with GCC on the POWER8, of 1.29x compared to the XL compiler. Interestingly, the compiler has achieved a better relative result on the POWER8 than it did on the Intel hardware when compared to the Intel compiler. It will be important future work to understand the root cause of this difference, and determine whether it can be easily optimised.

NVIDIA GPU Results: On the K20X, CCE suffered a 1.92x performance penalty compared to CUDA, while Clang suffered a 2.30x performance penalty, which is significantly less efficient than for the other mini-apps. As previously discussed, the `neutral` mini-app uses a single large computational kernel that requires many registers. When compiling CUDA for the P100, ptxas recognised 79 registers for CUDA, and when executing the OpenMP 4 code compiled with Clang, nvprof showed that 224 registers were used during the execution of the main computational kernel. This means that CUDA was 2.9x more efficient allocating registers, which is a considerable difference that we expect would be even worse for production applications with extensive Physics capabilities.

We could see that the occupancy achieved by CUDA was on average 37%, which is an acceptable level when targeting the P100 with this application. When compiled with Clang, the occupancy drops to 12%, demonstrating that CUDA is achieving 3.08x the occupancy on the P100, which likely explains the majority of

the 3.19x difference in overall runtime. The best performance that we achieved with Clang was with the number of registers restricted to 128 for both GPUs, which was the default behaviour for CCE.

5.4 Performance Discussion

The productivity and portability of OpenMP were highly consistent between the mini-apps, however, the performance was not. By far the most consistent was `hot`, which is intuitive as it is the simplest application, and hides many implementation inefficiencies behind long streaming data accesses. The `flow` mini-app exposed increased complexity, which meant that there was greater variation in the performance between the implementations. Also, it demonstrated that if you are not able to achieve vectorisation on the KNL, you may encounter significant performance issues, and the standard techniques failed for GCC in this case. The CCE and Clang compilers achieved impressive consistency for both applications on the NVIDIA GPUs, and we expect maturity to improve this even further in the future.

Although `neutral` has fewer kernels than the other two mini-apps, that kernel is long and complex, and the implementations performed significantly less consistently as a consequence. All architectures suffered from high variations in performance, and some did not emit hardware atomics correctly. On the GPU the primary issue was register pressure, which is actually quite a positive outcome, as we feel that this issue should be resolvable in the long term, and we expect that the implementations will be able to deliver significantly better performance relative to CUDA in the near future.

6 Future Work

It will be important to continue this research in the future to include new OpenMP 4 compilers, as well as tracking improvements to the existing implementations. As the *arch* project expands it will be insightful to extend the research to consider diverse applications, for instance stressing the task parallel features of the specification.

7 Related Work

An annual hackathon event for the improvement of OpenMP is hosted by IBM, where a live porting exercise is performed involving multiple US labs [11]. As an outcome of the 2015 hackathon, Karlin et al. [12] ported the applications Kripke, Cardioid, and LULESH to OpenMP, demonstrating some issues with the interoperability with some features of C++, and achieving performance with LULESH within 10% of an equivalent CUDA port.

There are many examples of studies that have looked at performance, portability and productivity of OpenMP with one or more applications [2,13–16], generally demonstrating that OpenMP is capable of highly productive application porting and high performance tuning. Other important studies have looked at the differences between OpenMP and other parallel programming models [6,17]. Lopez et al. [18], for instance, explored the ability for OpenMP 4.5 *and* OpenACC to achieve performance portability, and demonstrated success with multiple test cases, showing only minor performance differences between the two directive-based models.

8 Conclusion

The OpenMP specification has been designed to provide performance and portability, with some productivity enhancements compared to other models. Through this research we have discovered that performance and portability are certainly possible to an extent, but some aspects are limited by the specification and differences between compiler implementations. For instance, there have been few improvements to the standardisation of compiler implementations, which continues to preclude the writing of a single homogeneous directive to target multiple heterogeneous devices.

While porting a number of applications, we found that coupling data allocation with moving data onto the device reduces the amount of duplicate code, and ensures that data sizes are kept consistent, reducing bugs and increasing productivity. Also, relying on the implicit behaviour of OpenMP 4.0 for mapping scalar variables into a `target` region, as `map(tofrom: scalar-variable)`, can result in bugs when moving to OpenMP 4.5 compliant compilers.

Application developers who are used to the consistent performance delivered by the mature OpenMP implementations targeting CPU need to be aware that the implementations targeting other architectures are less mature. We even demonstrated that performance on the IBM POWER8 CPU is not necessarily consistent between implementations, likely due to maturity as well. This does not mean that the specification is not capable of enabling high performance on those architectures, but that the compiler implementations need time to improve.

The performance results we observed on modern HPC devices showed that, for applications with memory bandwidth bound kernels, OpenMP could generally achieve within 20–30% of the best possible performance. For the latency-bound application, the overheads introduced by the OpenMP implementations had a significant impact when offloading, and more variance was seen between compilers. Register pressure posed a significant issue for the `neutral` mini-app, which is something we expected from previous research efforts. It is not yet clear how to resolve the register issues, but it will be an important step towards achieving optimal OpenMP implementations.

Acknowledgements. The authors would like to thank the EPSRC for funding this research. We would also like to thank the Intel Parallel Computing Center (IPCC) at the University of Bristol for access to Intel hardware, and the EPSRC GW4 Tier 2 Isambard service for access to phase 1 of the Isambard supercomputer.

References

1. Heroux, M., Doerfler. D., et al.: Improving performance via mini-applications, Sandia National Laboratories, Technical report SAND2009-5574 (2009)
2. Martineau, M., McIntosh-Smith, S., Gaudin, W.: Evaluating OpenMP 4.0's effectiveness as a heterogeneous parallel programming model. In: Proceedings of 21st International Workship on High-Level Parallel Programming Models and Supportive Environments, HIPS 2016 (2016)
3. Eichenberger, A.E., et al.: OMPT: An OpenMP tools application programming interface for performance analysis. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 171–185. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40698-0_13
4. Antao, S.F., Bataev, A., Jacob, A.C., Bercea, G.T., Eichenberger, A.E., Rokos, G., Martineau, M., Jin, T., Ozen, G., Sura, Z., Chen, T., Sung, H., Bertolli, C., O'Brien, K.: Offloading support for OpenMP in Clang and LLVM. In: Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC, LLVM-HPC 2016, Piscataway, NJ, USA, pp. 1–11. IEEE Press (2016). https://doi.org/10.1109/LLVM-HPC.2016.6
5. Mellor-Crummey, J., Missing pieces in the OpenMP ecosystem. In: Keynote at International Workshop on OpenMP (2015)
6. Martineau, M., McIntosh-Smith, S., Boulton, M., Gaudin, W.: An evaluation of emerging many-core parallel programming models. In: Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM 2016 (2016)
7. Deakin, T., Price, J., et al.: BabelStream (UoB HPC Group) (2017). https://github.com/UoB-HPC/BabelStream
8. Lewis, E., Miller, W.: Computational Methods of Neutron Transport. Wiley, New York (1984)
9. Gentile, N.: Monte Carlo Particle Transport: Algorithm and Performance Overview. Lawrence Livermore, Livermore (2005)
10. Salmon, J.K., Moraes, M.A., Dror, R.O., Shaw, D.E.: Parallel randomnumbers: as easy as 1, 2, 3. In: 2011 International Conference for High Performance Computing, Networking, Storageand Analysis (SC), pp. 1–12. IEEE (2011)
11. Draeger, E.W., Karlin, I., Scogland, T., Richards, D., Glosli, J., Jones, H., Poliakoff, D., Kunen, A.: OpenMP 4.5 IBM November 2015 Hackathon: current status and lessons learned, Technical report LLNL-TR-680824, Lawrence Livermore National Laboratory, Technical report (2016)
12. Karlin, I., et al.: Early experiences porting three applications to OpenMP 4.5. In: Maruyama, N., Supinski, B.R., Wahib, M. (eds.) IWOMP 2016. LNCS, vol. 9903, pp. 281–292. Springer, Cham (2016). doi:10.1007/978-3-319-45550-1_20
13. Bercea, G., Bertolli, C., Antao, S., Jacob, A., et al.: Performance analysis of OpenMP on a GPU using a coral proxy application. In: Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems, p. 2. ACM (2015)

14. Lin, P.-H., Liao, C., Quinlan, D.J., Guzik, S.: Experiences of using the OpenMP accelerator model to Port DOE stencil applications. In: Terboven, C., Supinski, B.R., Reble, P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2015. LNCS, vol. 9342, pp. 45–59. Springer, Cham (2015). doi:10.1007/978-3-319-24595-9_4
15. Bertolli, C., Antao, S., Bercea, G.-T., et al.: Integrating GPU support for OpenMP offloading Directives into Clang. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015 (2015)
16. Hart, A.: First experiences porting a parallel application to a hybrid supercomputer with OpenMP4.0 device constructs. In: Terboven, C., Supinski, B.R., Reble, P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2015. LNCS, vol. 9342, pp. 73–85. Springer, Cham (2015). doi:10.1007/978-3-319-24595-9_6
17. Wienke, S., Terboven, C., Beyer, J.C., Müller, M.S.: A pattern-based comparison of OpenACC and OpenMP for accelerator computing. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014. LNCS, vol. 8632, pp. 812–823. Springer, Cham (2014). doi:10.1007/978-3-319-09873-9_68
18. Lopez, M.G., Larrea, V.V., Joubert, W., Hernandez, O., Haidar, A., Tomov, S., Dongarra, J.: Towards achieving performance portability using directives for accelerators. In: Proceedings of the Third International Workshop on Accelerator Programming Using Directives, WACCPD, 162016 (2016)