Martineau, M., McIntosh-Smith, S., & Gaudin, W. (2016). Evaluating OpenMP 4.0's Effectiveness as a Heterogeneous Parallel Programming Model. Paper presented at 21ST International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), Chicago, United States.DOI: 10.1109/IPDPSW.2016.70

Evaluating OpenMP 4.0's Effectiveness as a Heterogeneous Parallel Programming Model

Matt Martineau
HPC Group, University of Bristol
Bristol, United Kingdom
m.martineau@bristol.ac.uk

Simon McIntosh-Smith
HPC Group, University of Bristol
Bristol, United Kingdom
cssnmis@bristol.ac.uk

Wayne Gaudin
Atomic Weapons Establishment
Aldermaston, United Kingdom
wayne.gaudin@awe.co.uk

***Abstract*—Although the OpenMP 4.0 standard has been available since 2013, support for GPUs has been absent up until very recently, with only a handful of experimental compilers available. In this work we evaluate the performance of Cray's new NVIDIA GPU targeting implementation of OpenMP 4.0, with the mini-apps TeaLeaf, CloverLeaf and BUDE. We successfully port each of the applications, using a simple and consistent design throughout, and achieve performance on an NVIDIA K20X that is comparable to Cray's OpenACC in all cases. BUDE, a compute bound code, required 2.2x the runtime of an equivalently optimised CUDA code, which we believe is caused by an inflated frequency of control flow operations and less efficient arithmetic optimisation. Impressively, both TeaLeaf and CloverLeaf, memory bandwidth bound codes, only required 1.3x the runtime of hand-optimised CUDA implementations. Overall, we find that OpenMP 4.0 is a highly usable open standard capable of performant heterogeneous execution, making it a promising option for scientific application developers.**

***Keywords*-high performance computing; parallel computing; application programming interfaces; OpenMP; performance portability**

I. Introduction

Many of the world's largest supercomputing facilities either host heterogeneous architectures already, or plan to in the near future. Many-core devices, in particular, are seeing a surge in popularity because they boast high peak floating point and high memory bandwidth, while generally being more power efficient than traditional CPUs. Also, by design they are suited to processing large regular sets of data, which is a common requirement of modern scientific applications. Such devices are typically more complicated to develop for than when targeting CPUs, meaning that there is a strong demand for consensus about the programming models that should be used to target such devices [1].

The OpenMP 4.0 standard has introduced a number of new directives designed to support heterogeneous computational offloading in order to target many-core devices [2]. The introduction of such features into this prominent open standard demonstrates that there is an ever-increasing acceptance that such architectures will become a permanent feature in modern supercomputing. Although the specification has been in existence since the middle of 2013, compiler support for the heterogeneous features has been limited to a number of experimental open source implementations until more recently [3], [4], [5]. Until now, the principal use of OpenMP 4.0 has been for targeting the Intel Xeon Phi Knights Corner (KNC) architecture, but future releases of the Intel Xeon Phi architecture, such as the Knights Landing, are going to self-host, removing the requirement for an offloading model. In spite of this, the market for GPU co-processing is growing rapidly, and there is a compelling need for open, cross-platform programming models, shielding application developers from using non-portable, low-level APIs such as CUDA.

Prior research conducted by our group has shown that the open standard, OpenCL, can be used to develop performance portable HPC applications [6], [7]. We have also seen that such low-level APIs expose significant levels of complexity that may be off-putting for some application developers, and as such recognise the demand for an open standard that can provide performance portability with a lower barrier to entry. Recently, we have shown that directive-based parallel programming models can provide a highly usable interface while balancing good performance, generally within 20% of optimised low-level code, and significantly reduced development complexity [8]. In particular, we demonstrated that OpenMP 4.0 can perform well on CPUs and KNCs, and OpenACC can perform well on NVIDIA GPUs. This paper aims to show that OpenMP 4.0 is now capable achieving good performance on NVIDIA GPUs, making it a promising option for developers who want to target heterogeneous architectures, without committing to the potential development costs that come with other approaches.

II. The OpenMP 4.0 Standard

OpenMP is undoubtedly one of the most adopted parallel programming models, and represents a highly usable interface for targeting multi-core CPU architectures. We present the key new features of version 4.0 of the standard [2] relevant to our porting exercises.

A. Offloading to a Device

The great majority of the changes in the OpenMP 4.0 standard specifically support offloading some amount of computation to a target device. The most fundamental of those is the introduction of the **target** directive, which denotes a region of a code that will be directly mapped onto the device for execution. This directive exposes an important divergence between the two many-core targets, the GPU and KNC, where performant KNC codes can be written that include instructions inside **target** regions that should not be issued to a GPU.

This simple fact has an unfortunate consequence that codes written to target KNC today can quite easily be developed such that they would be reasonably challenging to adapt to also target GPUs. For instance, on KNC, we have found that reducing the number of **target** regions can improve performance, as they incur significant implicit synchronisation overheads, but might require extending the scope of the regions such that they include instructions that should be performed on the host for portability. Importantly, the **target** synchronisation overhead will not be an issue when targeting NVIDIA GPUs with OpenMP 4.0, as the compiler implementations can leverage the performant asynchronous queues provided by the CUDA runtime.

B. Managing Data Transfer

The specification incorporates a new conceptual model of a *device data environment*, that distinguishes the memory space local to a host processor and a target device, which is necessary to support offloading. To transfer data to and from the device, there are several new directives:

- **target map(*direction*: variable[*begin*:*end*])** - this requests that data is moved in the requested direction at the beginning and end of the **target** offload scope. The supported directions are: *to, from, tofrom*, and *alloc*, and the integral values *begin* and *end* denote the bounds of an array section.
- **target data map** - this expresses a data mapping scope that is free of any particular target regions, allowing data to be loaded onto the device prior to multiple target offload sections, potentially greatly reducing the number of data transactions. OpenMP 4.0 limits this to a structured lexical scope, but version 4.5 does not.
- **target update to / from** - this directive will immediately copy an array section **from** the device to the host or **to** the device from the host.

Another key directive included in the specification is the **target** conditional clause, **if(*condition*)**, which allows **target** regions to deactivate depending upon the supplied *condition*. This is an important mechanism to handle situations where functions are not always offloaded during execution, so that updates are made in the context of the correct data environment.

```
#pragma omp target data map(to: b[:n])
{
    // 'b' copied onto device

    // This region is offloaded to device
    #pragma omp target map(tofrom: a[:n])
    {
        // 'a' copied onto device
        #pragma omp teams distribute
        for(int ii = 0; ii < n; ++ii)
        {
            a[ii] = a[ii] + alpha * b[ii];
        }
    }
    // 'a' copied back

    // ... potentially more target regions
}
// 'b' is discarded by device
```

Figure 1. Code snippet showing the data movement and computational offloading in the context of two variables, 'a' and 'b'.

C. Multiple Levels of Parallelism

The OpenMP 4.0 standard introduces additional levels of parallelism with the **teams** and **distribute** directives, which respectively allow the developer to describe a league of thread teams, and then distribute them across the iterations of a loop. Depending on the target, this could place threads on the cores of a CPU, or block them onto the streaming multiprocessors of a GPU.

Further to the thread-level parallelism exposed through these new directives and the original **parallel for** clauses, the new standard introduces the **simd** directive. This essentially tells the compiler that a particular loop has independent iterations that can be concurrently executed using SIMD instructions, and is particularly useful for enabling vectorisation with minimal changes to a function. It is important to note, however, that it is sometimes still necessary to use directives like **ivdep** to ignore dependencies, in particular when the compiler believes there are output dependencies.

Each level of parallelism can be parameterised to some extent, where you can stipulate the number of teams (**num_teams**), threads (**thread_limit** or **num_threads**), vector lengths (**safe_len**), adjust the size of the iteration space (**collapse**), and alter the thread distribution schedule (**schedule** and **dist_schedule**). Ideally, compilers would be able to choose optimal default values for those parameters depending on the target, but it is likely that this parameterisation will become the key point for fine-tuning intra and inter-vendor performance across heterogeneous devices.

III. OpenACC and CUDA

OpenACC drew upon early ideas from the OpenMP accelerator subcommittee and combined them with vendor-specific efforts, enabling the first commercial compiler sup-

port to be introduced in 2012. An in-depth investigation performed by Wienke et al. [9] analysed the differences between OpenACC and OpenMP 4.0 and suggested that, while they are similar, OpenACC was slightly ahead of OpenMP 4.0 in terms of features. For instance, OpenACC 2.0 offers the **tile** and **cache** directives for optimisations, which are not present in OpenMP 4.0, however it does not include support for tasks or a mature implementation for targeting the CPU. We expect the differences between the standards will reduce over time, and later discuss some of the features introduced in version 4.5 of the OpenMP specification.

Given how new the Cray Compiling Environment (CCE) OpenMP 4.0 implementation is, we expected that the implementation would perform poorly in comparison to OpenACC. As part of our evaluation, we include performance results for OpenACC to gauge a loose upper bound for the performance that OpenMP 4.0 might be able to compete with in the future. Although OpenACC already provides good performance on GPUS, many users have existing codebases targeting the CPU that are written using OpenMP, and we expect that OpenMP 4.x is a good option for future-proofing investment, given that it builds upon an open standard with extensive vendor support.

CUDA was the first programming model capable of offloading compute to NVIDIA GPUs, and now encompasses an entire platform of technologies for this purpose [10]. We have included results of hand-optimised CUDA implementations of the mini-apps to represent a tight upper bound on the performance achievable by the directive-based implementations, as computational offloading to NVIDIA GPUs can only be achieved by generating instructions that are consumed by the CUDA runtime.

IV. Mini-Apps

In order to evaluate the performance attainable with the CCE implementation of the OpenMP 4.0 standard, we have ported three mini-apps. Mini-apps are miniaturised applications that represent the performance profile of genuine scientific workloads and algorithms, and they are intended to enable agile experimentation and benchmarking without having to alter production scientific codes.

In particular, our porting exercise focused on three mini-apps: TeaLeaf, CloverLeaf, and the BUDE benchmark. Each of the applications exposes a different set of programming requirements, and performance characteristics, and have been chosen specifically because they represent an important cross-section of modern scientific applications.

A. TeaLeaf: Implicit Heat-Conduction

TeaLeaf implicitly solves the heat-conduction equation using a number of matrix-free linear solvers: Conjugate Gradient (CG), Chebyshev and Preconditioned Polynomial CG (PPCG) [8]. The equation is solved over a spatially decomposed grid, with cell-centred temperatures and face-centred average densities. The structured grid can be decomposed and distributed to processing elements, using halo exchanges of a ghost-cell region to handle inter-process dependencies. Prior research of our group has demonstrated that this application can strong-scale and weak-scale well, particularly using the PPCG solver [11].

Importantly, TeaLeaf is a memory bandwidth bound code, and primarily represents the *Sparse Linear Algebra* dwarf of High Performance Computing [12], making it likely that the results are applicable to many other applications.

B. CloverLeaf: Explicit Hydrodynamics

CloverLeaf is an explicit Lagrangian-Eulerian hydrodynamics application that solves Euler's compressible fluid dynamics equations, which conserve mass, energy and momentum [13]. The application uses an explicit finite-volume approach, to second order accuracy, and time-marches across a staggered grid. The grid can be iterated in any order, while being passed through fourteen separate kernels, where each forward step in time (1) allows the cells to deform as nodes move into irregular spatial locations, and (2) advects the nodes back to their original locations, calculating the flux through each cell.

The mini-app has undergone extensive performance analysis, using a range of parallel programming models, and has been shown to weak and strong-scale effectively up to a high node count [13]. The application is memory bandwidth bound, and has a grid that is decomposed and distributed to individual processing elements, with halo exchanges between dependent patches, and as such represents a *Structured Grid* dwarf application [12].

C. Bristol University Docking Engine (BUDE) Benchmark

The BUDE benchmark uses a genetic algorithm to minimise the binding energy between two molecules, a receptor and ligand, by searching through successive generations of poses mutated from the best poses of the prior iteration [6]. The algorithm is compute-bound and comprised of a single large kernel with multiple nested loops, that is an example of both the *Monte Carlo* and *N-Body* dwarfs [12], making it quite distinct from TeaLeaf and CloverLeaf.

On top of a high frequency of arithmetic operations, the application requires multiple calls to math functions, and has a small memory footprint. Another important characteristic of BUDE is that it exposes the potential for tuning multiple parameters for performance on diverse devices, which we will perform for each port independently.

V. Development Findings

The development process uncovered several insights into the key issues that developers might face when developing OpenMP 4.0 applications targeting heterogeneous architectures. It is important to note that the following discussion

specifically relates to our experience using the CCE implementation of OpenMP 4.0, and that it is a very new implementation that will likely change and improve over time. We also expect that there could be some divergence between the approaches taken by different compiler vendors in the long-term.

A. Development Complexity

Our impression of OpenMP 4.0 so far is that it is expressive without exposing the levels of complexity seen with CUDA and OpenCL. In particular, we have found that it is relatively straightforward to describe parallelism within an application, while only requiring basic consideration for the separation between host and device memory spaces. It is important to note, however, that introducing OpenMP 4.0 into a legacy codebase will require more effort than writing OpenMP to just target CPUs. In particular, there is additional complexity in describing the data mappings, which need careful consideration in order to achieve good performance.

Targeting a single architecture will be the most straightforward, choosing the correct parallelisation at the thread and vector level, and then minimising communication of data between data-environments. The subtle complexity we expect to reside within the development task is in ensuring that large codebases can achieve performance portability with a single codebase. As compilers improve their support for OpenMP 4.0, it is likely that they will become more proficient at obscuring this complexity from the developer, but in the short term this problem is likely to get worse before it is solved.

There are some other complexities that we have encountered while developing OpenMP 4.0 code, that are caused by limitations in the specification. As discussed by Hart [14], the **map** directive cannot currently handle deep copies of structures, meaning that the developer is responsible for providing individual pointers to arrays, which was quite involved in the case of our mini-app ports. However, a nice feature of the OpenMP 4.0 implementation is that once a **target data** region was set up above the main timestep loop, the mapping directives did not have to be repeated at the loop-level.

There is some ambiguity in the use of the **simd** directive, which the specification states should be placed above loops with independent iterations that can be converted into SIMD instructions. We are not sure how this fits into the GPU-targeting model where SIMD instructions are not necessarily available, but found that BUDE would only parallelise once the **simd** directive was included alongside the **teams distribute** above the main loop.

In spite of the additional considerations involved in using the standard's new features, we believe that application developers wanting to write or port applications to target heterogeneous architectures are likely to find OpenMP 4.0 hard to beat in terms of development cost.

B. Compiler-Specific Implementation Restrictions

Not only is the CCE implementation of OpenMP 4.0 new, it actually represents the first commercially supported implementation. The documentation includes some limitations and restrictions present in their compliance to the specification, but it is of course important to note that these are liable to change given that the implementation is under active development.

The documentation for CCE 8.4.3 [15] states that user-defined reductions are not supported within **target** regions, and this makes sense given the additional complexity inherent with GPU reductions. Further to this, there is some limitation on the set of OpenMP API calls that are allowed within a target region, with a key exclusion being **omp_get_max_threads**, which you would expect to provide the maximum number of threads allowed within a threadblock. We did not encounter any issues because of this limitation while porting the mini-apps, but it may be important for some cases of optimisation.

Another CCE restriction is that **parallel** statements are limited to a single thread when declared inside **target** regions. Unfortunately, we were not able to develop a code that successfully accelerated while including a **parallel** region, presumably because of an issue with the implementation. The OpenMP 4.0 specification set out a restriction that threads other than the master thread of a team would only execute once a **parallel** region was encountered, but version 4.5 of the standard doesn't include this restriction. The CCE implementation adheres to version 4.5 in this case, allowing **for** loops to be decorated with a **target teams distribute** and no **parallel for**. This point is quite subtle, and caused no issues for our performance evaluation, but might have implications for the long-term performance portability of codes, where applications could be written biased towards GPUs, without consideration for **parallel** regions.

C. Outcomes

We had originally planned to include results for a port of another mini-app, which contains a sweep-style algorithm, but we were not able to successfully parallelise the main loop with the CCE implementation. The root cause of the problem stemmed from the use of indirection arrays for indexing inside the loop, and affected the CCE OpenACC implementation as well. The loop does parallelise with other programming models, including OpenACC with PGI, and so we expect the problem is resolvable, whether it lies in our implementation or CCE.

Porting TeaLeaf, CloverLeaf and the BUDE benchmark turned out to be reasonably straightforward and following optimisation, discussed in the next section, the final design for each port was very simple. The difference between the Intel and CCE interpretations of the OpenMP 4.0 specification meant that it was necessary to change the common set of directives at each loop, as shown in Figure 2.

```
// For KNC
#pragma omp target if(offload) device(dev_id)
#pragma omp parallel for simd reduction(+: temp)
for(int ii = 0; ii < y; ++ii)
{
    ...

// For GPU
#pragma omp target if(offload)
#pragma omp teams distribute reduction(+: temp)
for(int ii = 0; ii < y; ++ii)
{
    ...
```

Figure 2. Demonstrating the subtle difference between OpenMP 4.0 targeting KNC with Intel icc and NVIDIA GPUs with CCE.

Although the **device** clause could not be placed on the **target** regions, it can still be set using the OpenMP API calls, and so we cannot see any issues with this.

VI. Optimisations

In order to present the fairest results possible, we performed extensive performance optimisation of each application. Importantly, each of the applications already has multiple optimised versions for targeting different architectures, and we concentrated on tuning our new ports specifically for GPU and CPU, where relevant. Of course, there is less opportunity for optimisation with the high-level directive-based models than with low-level APIs such as CUDA or OpenCL, but it was definitely possible to support the compiler to generate efficient code.

A. The Three Pillars of Many-Core Optimisation

When programming for GPUs, we believe that the best performance gains can be achieved with the three pillars of many-core optimisation:

- **Minimise Data Communication:** Communicating across PCI-express is very expensive and should be minimised as a priority, where the aim is generally to persist as much data as possible on the target device and only synchronise data when strictly necessary.
- **Maximise Utilisation:** The parallelisation scheme for each performance critical loop should be considered to ensure that the iteration space is large enough to saturate the target device with work.
- **Chase the Bound:** In the case of compute-bound functions, this entails reducing the number of arithmetic operations and utilising features such as fast intrinsic maths and Fused-Multiply-Add wherever possible. For memory-bound functions, the frequency of non-cached loads and stores must be minimised, coalescence enabled with contiguous memory layouts, and the number of non-redundant arithmetic operations maximised to support memory access latency hiding, perhaps through kernel fusion.

Although OpenMP 4.0 appears to be less flexible than APIs like CUDA and OpenCL, it is still possible to perform significant optimisations tackling each of the three key pillars.

B. Experience

Prior to optimisation, data was copied to and from the device for each **target** offload region, and the applications were an order of magnitude slower than the best-case CUDA implementations. However, for all applications it was possible to place **target data** regions surrounding the main timestep loops, persisting the majority of data on the device for the extent of the solve without synchronisation, minimising the amount of expensive data transfer. This optimisation has by far the biggest impact on performance and brought the applications significantly closer to the performance of our CUDA implementations.

In order to ensure the device is saturated with work, several techniques exist, such as re-arranging the parallelisation schemes, fusing kernels, and collapsing nested loops to increase the iteration space. For TeaLeaf and CloverLeaf, significant work is exposed at the loop-level and collapsing the nested loops proved detrimental to performance, so we advise careful consideration is given to whether collapsing is required when porting such codes. We are not currently sure whether loop collapsing will be necessary between devices, or whether compilers will be able to perform this optimisation automatically. If collapsing cannot be handled satisfactorily by the compiler, it could become an important parameter for heterogeneous performance portability.

The chosen mini-apps are already highly optimised in terms of their memory access patterns, and so no additional memory optimisation was required. In general, this step would include fusing kernels and recognising redundant memory accesses at the loop-level. Another concern might be developing optimal reductions, a reasonably complicated albeit re-usable step, but with OpenMP 4.0 this problem can be deferred to the compiler. We did consider that scheduling the threads correctly could improve the potential for memory coalescence, and hypothesised that utilising **dist_schedule(static,1)** would lay out the threads contiguously to support this. Upon introducing the scheduling to some critical loops we saw no change in performance, which suggests that this layout is being chosen by the compiler as a default.

The BUDE benchmark includes a higher ratio of arithmetic operations than the other applications, and uses an optimisation technique where work can be oversubscribed to threads, in order to exploit some repetitive computations. As a consequence, it was necessary to restrict the number of registers used by passing the CUDA compiler parameter *--maxrregcount* via the Cray compiler, to avoid reducing potential occupancy. We also found that the CCE OpenMP 4.0

implementation achieved marginally better performance with **for** loops rather than equivalent **while** loops.

VII. Performance Results

In order to demonstrate the potential performance that can be attained with the CCE GPU implementation of OpenMP 4.0, we have conducted a number of performance evaluations. All of the testing is performed on Cray Inc.'s XC40 supercomputer, Swan, and uses the following devices:

- **Intel Xeon Haswell CPU (E5-2698 v3 @ 2.30 GHz):** We target a single socket (16 cores) using optimised OpenMP code compiled with both CCE 8.4.3 and Intel 15.0.3.
- **NVIDIA K20X GPU:** We target a single GPU as this is sufficient to demonstrate how well the OpenMP 4.0 GPU-targeting offload code performs. Our results include CUDA (version 7.0), OpenACC (PGI 15.10 and CCE 8.4.3) and OpenMP 4.0 (CCE 8.4.3).

From our perspective, the devices chosen represent popular modern HPC devices, and should offer a representative view of the performance of each model.

A. CloverLeaf

The CloverLeaf mini-app was fully parallelised using OpenMP 4.0 and OpenACC, and optimised as far as possible for the available compilers. We have chosen to use a problem size of 3840^2 for 87 steps, because it represents a large data processing task that is particularly suited to GPUs.

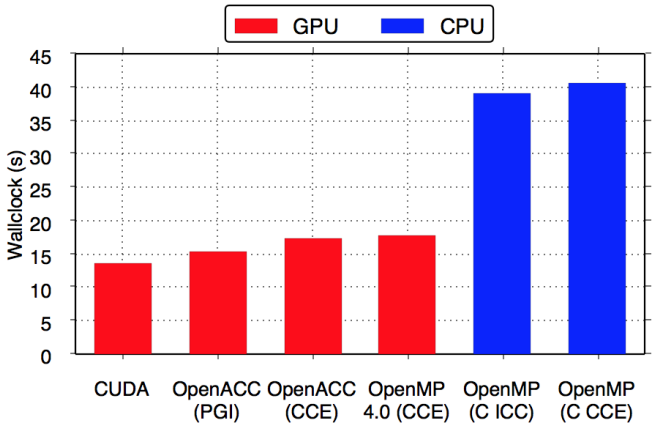

Figure 3. Results for CloverLeaf - 87 steps of 3840^2 problem.

When comparing the CPU and GPU results, shown in Figure 3, it is evident that all of the GPU results are at least 2x faster than the CPU for this particular problem. Further, the wallclock results demonstrate that OpenMP 4.0 offers nearly identical performance to the equivalently optimised OpenACC code compiled with CCE. This is an impressive result that exceeded our expectations, given that the CCE implementation of the OpenMP 4.0 specification is so new. Compared to the hand-optimised CUDA implementation, the OpenMP 4.0 port only required 1.3x the runtime, which represents a highly competitive result given the reduced development complexity.

Interestingly, we observe a significant difference between the OpenACC results, where CCE is 1.3x slower than CUDA, and PGI is only 1.15x slower. The 15% performance difference compared to CUDA is negligible, and demonstrates the extent to which OpenMP 4.0 could be optimised in the future.

B. BUDE

The BUDE benchmark was parallelised with OpenMP 4.0 and OpenACC, however, when compiling with PGI the port output incorrect checking values, and so PGI results have been excluded. In all performance experimentation we chose a problem size of N=57344, which was determined as an ideal problem size for the K20X GPU because it is a multiple of the 14 compute units, running 128 wide threadblocks.

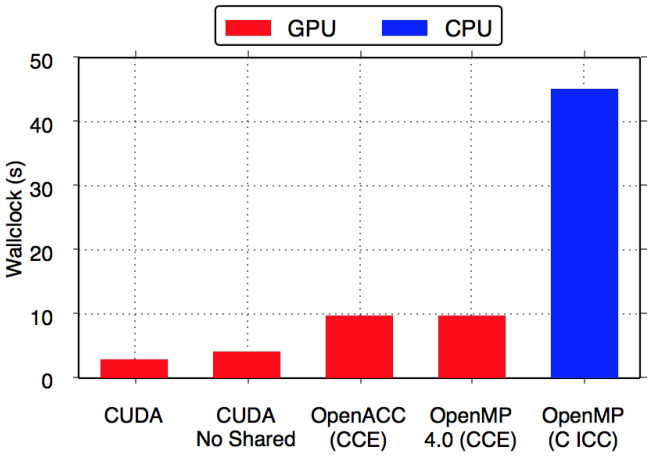

Figure 4. Results for the BUDE benchmark - 100 iterations of $N = 57344$ problem.

The results in Figure 4 show that the CPU implementation required roughly 18x the runtime of our most optimised CUDA implementation, while the OpenMP 4.0 implementation was able to achieve identical performance to the OpenACC port. The CUDA (no shared) port represents the performance attained without a specific optimisation, where some memory from DRAM is transferred to shared memory, overlapping the copy with computation. The OpenMP 4.0 implementation requires 2.2x the runtime of this version, which is significantly higher than seen with the other applications, but still good performance given the simplicity of the port. The most optimised CUDA port increases this gap to nearly 3.5x, and we think that this problem exposes a significant limitation of the existing specification. We propose that it would be beneficial to incorporate an independent directive to copy an array section from DRAM into shared memory, restricting synchronisation to the latest possible

opportunity. OpenACC 2.0 provides the **cache** directive, which we expected could provide this functionality, but is restricted to innermost loops meaning it was not capable of performing this particular optimisation.

C. TeaLeaf

As with CloverLeaf, the TeaLeaf mini-app was successfully parallelised with OpenMP 4.0 and OpenACC, and this includes the three main solvers offered by the application. The chosen 4096^2 problem represents the point of mesh convergence and therefore the largest problem that we could choose to evaluate performance on the GPU. We include results for all three of the solvers as this extends the range of different linear algebra kernels that are offloaded.

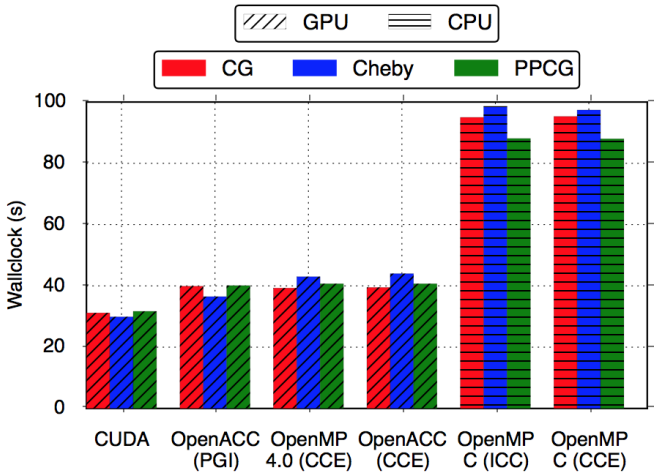

Figure 5. Results for TeaLeaf - single step of 4096^2 problem.

Figure 5 demonstrates that, as with CloverLeaf, the performance on the GPU is at least 2x better than the CPU, giving us greater confidence in the overall result. We can also see that CCE's OpenACC and OpenMP 4.0 exhibit identical performance. This would seem to suggest that the actual code generated by the two implementations could even be identical, however analysis of the PTX outputs suggested that this was not the case. The OpenACC code compiled with PGI has a faster Chebyshev solver, and we have not been able to determine why this occurs, or achieve the same level of performance for the CG and PPCG solvers.

Most importantly, the OpenMP 4.0 port is on average only 1.3x slower than the hand optimised CUDA code, which again demonstrates impressive performance for the low cost of development. Interestingly, in previous research we have seen some instability in the results between solvers for TeaLeaf [8], however the results here are quite stable, which we believe is a reasonable predictor that other linear algebra codes could achieve similar results.

VIII. Analysis

So that we can understand the causes and implications of the results, we present some analysis of the different ports and our expectations of the standard's ability to achieve performance portability.

A. Analysis of the Generated PTX

It was not possible to collect exact statistics through manual analysis of the PTX outputs because of their extensive use of conditional blocks. However, we did observe that the OpenMP 4.0 PTX outputs for each of the applications contained significantly more lines of code than the CUDA PTX outputs, and appeared to contain a higher frequency of arithmetic and control flow instructions. We expected that the memory-bound application's outputs would suffer from inflated loads and stores, but this was not the case.

B. NVIDIA Profiler (nvprof)

As the CCE implementations of OpenMP 4.0 and OpenACC are built on top of the CUDA platform, they can be profiled using the NVIDIA profiler **nvprof**, which enables the collection of an extensive set of performance counters from the GPU. We present pertinent counters relevant to each application, in order to uncover potential reasons for their different performance profiles.

Table I
PROFILING STATISTICS FOR THE **CALCULATE U AND R** KERNEL OF THE TEALEAF CG SOLVER.

Type	**CUDA**	**OpenACC**	**OpenMP**
Multiprocessor Activity (%)	99.8	99.4	99.4
Instructions Per Cycle	1.06	0.37	0.39
G.Memory Replay Overhead (%)	5	19	18
Control Flow Inst. (mil.)	5.0	4.7	4.8
Floating Point Ops. (mil.)	100	102	102
DRAM Read Transactions (mil.)	20.0	22.4	22.4
DRAM Write Transactions (mil.)	10.1	11.7	11.7
Runtime(s)	**14.0**	**17.7**	**17.8**

1) TeaLeaf Profiling Metrics: In Table I we demonstrate some of the key profiling results that we obtained for this analysis. Notably, OpenMP 4.0 and OpenACC have almost identical performance metrics, indicating the similarities in the two implementations. In all cases the multiprocessor activity demonstrates full utilisation of the streaming multiprocessors on the device, and supports our earlier observation that the **collapse** statement was unnecessary because of the amount of work exposed by the applications. The instructions per cycle shows a significant divergence between the models, with OpenACC and OpenMP 4.0 achieving only 40% compared to CUDA, which we believe is indicative of increased impact of memory latencies, potentially correlated by the increased overhead of global memory cache-miss replays.

The number of floating point operations is very consistent between the three models, which is unsurprising given that the kernel contains only 6 arithmetic operations in total. The access to DRAM, which is typically the major bottleneck for memory bandwidth bound codes, was significantly different between the models, where the number of read and write transactions are 12% and 16% higher than CUDA, respectively. We expect that this increased memory requirement can explain a significant portion of the 27% additional runtime, but could not uncover the reasons for this additional DRAM utilisation.

Table II
PROFILING STATISTICS FOR THE SET OF **CELL ADVECTION** KERNELS IN CLOVERLEAF.

Type	**CUDA**	**OpenACC**	**OpenMP**
Multiprocessor Activity (%)	99.8	99.8	99.7
Instructions Per Cycle	1.72	1.55	0.99
Control Flow Inst. (mil.)	59.5	68.6	67.1
Floating Point Ops. (bil.)	2.5	3.5	3.5
DRAM Read Transactions (mil.)	112	116	133
DRAM Write Transactions (mil.)	53	54	60
Runtime(s)	**2.4**	**2.8**	**3.5**

2) CloverLeaf Profiling Metrics: Table II demonstrates a quite different scenario to the one seen with TeaLeaf, where the runtime is markedly different between the three implementations. Again, the multiprocessor activity is at maximum, which gives confidence in the quantity of work available to the GPU, but this is the only consistent metric.

The kernels are far more complicated and arithmetically intensive than the kernel analysed for TeaLeaf, and we can see that the amount of control flow instructions and floating point operations is much higher in both OpenACC and OpenMP 4.0. This suggests that there is some inefficiency in the code generation for both implementations, and that they likely follow similar arithmetic generation and optimisation paths.

The memory transactions for OpenACC are only around 2-4% higher than CUDA, which leads us to believe that the majority of the performance difference resides in the greater frequency of control flow and floating point operations. Most importantly we can see that, compared to OpenACC, OpenMP 4.0's read transactions are around 15% higher and write transactions are around 11% higher, likely accounting for a significant proportion of the 25% difference in runtime.

3) BUDE Profiling Metrics: In Table III, we compare the OpenMP 4.0 and OpenACC results to that of the CUDA (no shared) port (seen in Figure 4), so that the comparison is less skewed by the shared memory optimisation. We can see that there are some significant differences between the implementations, where they are only similar in their efficient utilisation of the available multiprocessors and the number of instructions executed per cycle.

Table III
PROFILING STATISTICS FOR THE BUDE COMPUTATION KERNEL.

Type	**CUDA**	**OpenACC**	**OpenMP**
Multiprocessor Activity (%)	97.1	99.9	99.9
Control Flow Inst. (mil.)	6	324	324
Floating Point Ops. (bil.)	35.3	43.4	43.4
Instructions Per Cycle	3.6	3.7	3.6
Instruction Issued (bil.)	1.7	4.3	4.3
Instruction Replay Overhead (%)	13	17	17
DRAM Read Transactions (mil.)	1.5	3.0	3.1
DRAM Write Transactions (mil.)	0.08	1.8	1.8
Runtime(s)	**4.4**	**9.7**	**9.8**

The number of control flow instructions executed is so divergent we did not initially believe the results, and still cannot find a reason as to why this number is so inflated, but we do expect it is having a significant impact on the performance of the ports. The volume of floating point operations is 23% higher for OpenACC and OpenMP 4.0, suggesting some inefficiency in the arithmetic optimisation. The number of instructions executed per cycle is consistent between the models, but the number of instructions issued is *significantly* higher for OpenMP 4.0 and OpenACC, which also suffer from a marginally higher proportion of replays.

The number of read transactions that had to be fulfilled from DRAM is 2x that of CUDA, and the number of writes is over 20x higher. While this difference is significant, the scheduling built into NVIDIA GPUs is likely to hide the performance degradation of the latency by overlapping the memory access with the dominant compute. We expect that the increased control flow, instruction count, and arithmetic intensity combined were responsible for the majority of the performance degradation, given that BUDE is a compute bound code.

C. Performance Portability

We have clearly shown that OpenMP 4.0 is now a competitive option for performance on CPUs and GPUs, and have demonstrated separately that it can achieve good performance offloading to the Knights Corner architecture [8]. It will be an important future concern to understand how performance will be achieved on all architectures from within a single code base.

Our research suggests that satisfactory performance portability will only be achievable for production codes if the programming model can support a single source representation of the domain logic. Further to this, it will be essential that portability work-arounds, such as adding conditional preprocessor directives, can be avoided or minimised in favour of expressing multiple parallelisation schemes at the loop level and allowing the compiler to select the optimal scheme depending on the architecture. The difference between those

two paradigms is subtle but should greatly reduce the need for codes to be re-written as target architectures evolve.

Throughout the development of the OpenMP 4.0 ports, we have been able to assess the potential for performance portability exposed by the framework. The current specification appears to expose the necessary loop-level control such that parallelism can be expressed for CPUs, GPUs and KNCs, and we expect that the directives necessary to target those architectures can co-exist at the loop-level. Essentially, it should be possible to place directives above all functional loops that parallelise the work into teams, threads and vectors, allowing the compiler to choose which are relevant and then parameterising the widths for fine-tuned optimisation.

D. OpenMP 4.5

It is important to recognise that the OpenMP standard is being actively improved, with a focus on heterogeneous architectures, and we now introduce some of the new features that are included in the recently released version 4.5 of the specification [16].

As previously mentioned, the mini-apps that we have chosen for this research are well structured and benefit from being able to maintain the majority of data on a target device for the full duration of the solve. However, there are applications where persisting data across a structured lexical scope may not be possible because of conditionality inherent in structural or controlling code, which could make it difficult or even impossible to persist the data optimally on the device for all branches. The new unstructured data regions included in OpenMP 4.5 support this conditionality with independent **target data enter** and **target data exit** clauses. This feature is going to be particularly important when introducing OpenMP 4.0 into large multi-functional applications containing complex controlling code.

While testing OpenMP 4.0 on the KNC architecture, we encountered significant overheads, which we hypothesised were caused by **target** regions having no queuing mechanism, and as such being offloaded one at a time, synchronising with the host at both ends of the call. The new specification introduces the **nowait** directive for **target** regions, which potentially allows multiple kernels to be queued onto a device, reducing the synchronisation overheads. Of course, this functionality is already exposed by the CUDA runtime and so does not have significant influence for NVIDIA GPU targets, but may be important for future performance portability.

Finally, we have encountered a situation where it is impossible to express the same reduction functionality using the OpenMP 4.0 specification as we had in CUDA. In particular, we use a mini-app that contains a function that loops over two arrays and accumulates multiple values into each element, which can be partially reduced using shared memory in CUDA, and that we believe can be expressed within OpenMP 4.5 using array sections inside the **reduction** directive.

The new features in version 4.5 are likely to take time to be implemented, given that there is a dearth of mainstream implementations of OpenMP 4.0, but these changes will be useful for large applications, and demonstrate an important commitment to the heterogeneous features of the standard.

IX. Related Work

Acknowledging the lack of OpenMP 4.0 implementations targeting GPUs, Liao et al [5] developed a prototype implementation using the ROSE compiler that could target NVIDIA GPUs, and Lin et al. [17] used that implementation to demonstrate that the model is capable of targeting accelerators. Ozen et al. [4] partially implemented OpenMP 4.0 in the OmpSs compiler and performed a performance evaluation with three kernels. Bertolli et al. [3] and Bercea et al. [18] implemented GPU support for Clang using the OpenMP 4.0 specification, and presented performance results for a representative set of kernels in LULESH.

McIntosh-Smith et al. [6], Martineau et al. [8] and Mallinson et al. [13] investigated the performance of the BUDE, TeaLeaf and CloverLeaf mini-apps, respectively, across multiple architectures and programming models. Hart [14] ported the NekBone mini-app to use OpenMP 4.0 and target GPUs via the Cray Compilation Environment, proposing some best practices for porting existing applications to use OpenMP 4.0. Dietrich et al. [19] implemented a performance measurement library that allowed measurement of OpenMP 4.0 code running on the KNC architecture. Wienke et al. [9] compared OpenMP 4.0 and OpenACC, predicting that OpenMP 4.0 would likely achieve best adoption in the long-term because it is such a prominent standard, and proposing performance evaluations as important future work.

X. Future Work

Throughout this investigation, it has become apparent that there is some ambiguity in OpenMP 4.0's support for expressing parallelism, for instance, where should **reduction**s be placed for performance portability? Importantly, this will have an influence on the ability for a particular code to compile with different compilers or target different architectures without code changes. As previously discussed, we believe that tuning the parameters exposed by OpenMP 4.0 may be necessary on a per-device basis to achieve the best possible performance, but hope to find a way that the directives can be written once and work on the majority of architectures. We propose this as an essential area for future research, as scientific codes being developed or ported *must* have flexibility to new high performance architecture.

It will also be important to track the improvement of the CCE implementation of OpenMP 4.0, and eventually investigate the performance achieved by different compilers.

XI. Conclusion

This research has utilised the CCE implementation of OpenMP 4.0 to gather preliminary performance results for mini-apps running on NVIDIA GPUs. All of the presented mini-apps were successfully ported, and achieved performance comparable to OpenACC, while requiring 2.2x the runtime of an equivalent CUDA implementation for the compute-bound code, BUDE, and 1.3x for the memory bandwidth bound codes, TeaLeaf and CloverLeaf. The gap between the OpenMP 4.0 and CUDA implementations of BUDE could be increased to over 3.5x if some data stored in DRAM was cached into shared memory and overlapped with compute, and we believe it would be useful if the OpenMP specification could be extended to support this optimisation.

We carefully analysed the performance profiles of each of the ports and inferred some potential causes of the performance difference, including increased DRAM access and floating point operations. While we do believe that there is room for improvement with the performance offered by the CCE implementation, the results are very promising, with OpenMP 4.0 now balancing performance and reduced development costs whilst targeting the most popular HPC devices. We conclude that, as compiler support improves, and awareness of the capabilities of OpenMP 4.x spreads, the model has the potential to become the defacto standard for targeting heterogeneous architectures.

Acknowledgment

This work was funded by an EPSRC CASE studentship supported by the UK Atomic Weapons Establishment (AWE). We would like to thank the AWE for their support of this research, and Cray Inc. for their support and provision of the XC40 supercomputer Swan.

References

[1] J. Dongarra *et al.*, "The International Exascale Software Project roadmap," *International Journal of High Performance Computing Applications*, 2011.

[2] OpenMP Architecture Review Board, "OpenMP Application Program Interface v4.0," 2013.

[3] C. Bertolli, S. Antao, G. Bercea *et al.*, "Integrating GPU support for OpenMP offloading directives into Clang," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. ACM, 2015, p. 5.

[4] E. L. J. Ozen, G. Ayguadé, "On the Roles of the Programmer, the Compiler and the Runtime System When Programming Accelerators in OpenMP," in *Using and Improving OpenMP for Devices, Tasks, and More*. Springer, 2014, pp. 215–229.

[5] C. Liao, Y. Yan, B. de Supinski *et al.*, "Early Experiences with the OpenMP Accelerator Model," in *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, 2013, pp. 84–98.

[6] S. McIntosh-Smith, J. Price, R. Sessions, and A. Ibarra, "High Performance in Silico Virtual Drug Screening on Many-Core Processors," *International Journal of High Performance Computing Applications*, pp. 119–134, 2014.

[7] S. McIntosh-Smith and D. Curran, "Evaluation of a Performance Portable Lattice Boltzmann Code Using OpenCL," in *Proceedings of the International Workshop on OpenCL 2013 & 2014*, ser. IWOCL '14. New York, NY, USA.: ACM, 2014, pp. 2:1–2:12.

[8] M. Martineau, S. McIntosh-Smith, M. Boulton, and W. Gaudin, "An Evaluation of Emerging Many-Core Parallel Programming Models," Accepted to 7th International Workshop on Programming Models and Applications for Multicores and Manycores, 2016.

[9] S. Wienke, C. Terboven, J. C. Beyer, and M. Müller, "A Pattern-Based Comparison of OpenACC and OpenMP for Accelerator Computing," in *Euro-Par 2014 Parallel Processing*. Springer, 2014, pp. 812–823.

[10] N. Corporation, "CUDA C Programming Guide v7.0," 2016.

[11] S. McIntosh-Smith, M. Boulton, W. Gaudin, and P. Garrett, "Optimising sparse iterative solvers for many–core computer architectures," Presentation at SUBWOG 2015, 2015.

[12] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis *et al.*, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.

[13] A. Mallinson, D. Beckingsale, W. Gaudin, J. Herdman, and S. Jarvis, "Towards Portable Performance for Explicit Hydrodynamics Codes," in *The International Workshop on OpenCL (IWOCL)*, 2013.

[14] A. Hart, "First Experiences Porting a Parallel Application to a Hybrid Supercomputer with OpenMP 4.0 Device Constructs," in *OpenMP: Heterogenous Execution and Data Movements: 11th International Workshop on OpenMP, IWOMP 2015, Proceedings*, 2015, pp. 73–85.

[15] Cray Inc., "C and C++ Reference Manual (S-2179-84)," 2016.

[16] OpenMP Architecture Review Board, "OpenMP Application Program Interface v4.5," 2015.

[17] P. Lin, C. Liao, D. Quinlan *et al.*, "Experiences of Using the OpenMP Accelerator Model to Port DOE Stencil Applications," in *OpenMP: Heterogenous Execution and Data Movements: 11th International Workshop on OpenMP, IWOMP 2015, Proceedings*, 2015, pp. 45–59.

[18] G. Bercea, C. Bertolli, S. Antao *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*. ACM, 2015, p. 2.

[19] R. Dietrich, F. Schmitt, A. Grund, and D. Schmidl, "Performance Measurement for the OpenMP 4.0 Offloading Model," in *Euro-Par 2014: Parallel Processing Workshops*. Springer, 2014, pp. 291–301.