2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems

Performance Analysis and Optimization of Clang's OpenMP 4.5 GPU Support

Matt Martineau, Simon McIntosh-Smith
University of Bristol, UK
Email: {m.martineau, cssnmis}@bristol.ac.uk

Carlo Bertolli, Arpith C. Jacob, Samuel F. Antao,
Alexandre Eichenberger, Gheorghe-Teodor Bercea,
Tong Chen, Tian Jin, Kevin O'Brien, Georgios Rokos,
Hyojin Sung, Zehra Sura
IBM T.J. Watson Research Laboratory, NY USA,
Email: {cbertol, acjacob, sfantao, alexe, gheorghe-teod, chentong, tijin, caomhin, grokos, hsung, zsura}@us.ibm.com

***Abstract*—The Clang implementation of OpenMP® 4.5 now provides full support for the specification, offering the only open source option for targeting NVIDIA® GPUs. While using OpenMP allows portability across different architectures, matching native CUDA® performance without major code restructuring is an open research issue.**

In order to analyze the current performance, we port a suite of representative benchmarks, and the mature mini-apps TeaLeaf, CloverLeaf, and SNAP to the Clang OpenMP 4.5 compiler. We then collect performance results for those ports, and their equivalent CUDA ports, on an NVIDIA Kepler GPU. Through manual analysis of the generated code, we are able to discover the root cause of the performance differences between OpenMP and CUDA.

A number of improvements can be made to the existing compiler implementation to enable performance that approaches that of hand-optimized CUDA. Our first observation was that the generated code did not use fused-multiply-add instructions, which was resolved using an existing flag. Next we saw that the compiler was not passing any loads through non-coherent cache, and added a new flag to the compiler to assist with this problem.

We then observed that the compiler partitioning of threads and teams could be improved upon for the majority of kernels, which guided work to ensure that the compiler can pick more optimal defaults. We uncovered a register allocation issue with the existing implementation that, when fixed alongside the other issues, enables performance that is close to CUDA.

Finally, we use some different kernels to emphasize that support for managing memory hierarchies needs to be introduced into the specification, and propose a simple option for programming shared caches.

I. Introduction

In this paper we present a performance analysis of a set of synthetic kernels and mini-apps using the Clang OpenMP 4.5 compiler. Our findings lay the initial foundations for implementing an optimizing compiler and runtime library for heterogeneous systems.

Two new supercomputers, Sierra and Summit, are nearing installation and will contain a mixture of IBM® POWER9 CPUs, and NVIDIA Volta GPUs [1]. Lawrence Livermore National Laboratory and Oak Ridge National Laboratory are procuring the clusters, but they will be also be used by Los Alomos National Laboratory and Sandia National Laboratory, who are procuring their own supercomputers with Intel® Xeon® and Xeon Phi™ processors. The diversity and heterogeneity of the architectures available in those supercomputers presents scientific application developers with a significant challenge in terms of portability and performance that needs to be addressed.

In order to take advantage of modern supercomputing resources, scientific application developers are going to have to make significant changes to their applications. Many of those applications are monolithic and rewriting them represents an onerous and extremely expensive task, for which repetition needs to be limited. Although the CUDA framework is well suited to targeting GPUs and offers a number of features for performance optimization: (1) it exposes the architecture directly to the user, which adds complexity for application developers, and (2) the resulting code will only be able to run on NVIDIA GPUs.

This has lead to a significant increase of interest in portable programming techniques and technologies, and a new wave of programming models to support this need. A number of options exist that can target heterogeneous architectures, for instance, OpenMP, OpenCL™, OpenACC™, RAJA and Kokkos [2] [3] [4] [5]. Each model adopts a different approach, and as such provide some trade-offs in terms of their productivity, complexity and performance. In particular, OpenMP provides a highly productive interface with minimal complexity, and benefits from widespread adoption within the scientific community, with the majority of HPC vendors supporting version 3.1 of the specification [6].

The Clang OpenMP 4.5 implementation represents the first open source OpenMP compiler for targeting NVIDIA GPUs, and is part of an industry-wide effort to support the next generation of supercomputers [7]. The implementation was written by the Advanced Compiler Technologies team at the IBM T.J. Watson Research Center, as part of the CORAL project, and is now feature-complete.

This implementation leverages the Clang compiler frontend, providing a runtime for execution on NVIDIA GPUs with CUDA, and is actively being introduced into the Clang compiler trunk. We aim to take this compiler implementation and determine what performance it can achieve relative to the best performance achievable with CUDA. We find a number

978-1-5090-5218-9/16 $31.00 © 2016 IEEE
DOI 10.1109/PMBS.2016.11

of insights that can provide compiler cost models to form an optimizing theory for code synthesis and runtime library development which will benefit all languages targeting GPUs.

II. Contributions

There are a number of contributions presented in this paper:

- We present results of micro-benchmarks and mini-apps that have been ported to CUDA and OpenMP 4.5.
- We discover optimizations that improve the performance of Clang's OpenMP 4.5 support, implementing a number of them and proposing how the rest can be incorporated into the compiler and specification.
- We present motivation for introducing a mechanism for exploiting shared memory with OpenMP and propose a potential approach for implementing this.

III. Background

OpenMP 4.0 introduced a number of features designed to support targeting heterogeneous architectures using an offloading model [8]. OpenMP 4.5 extended this functionality with a number of important improvements, for instance: all variables default to **firstprivate**, allowing them to be passed as parameters to CUDA kernels, and the **target data enter** and **target data exit** directives were introduced to enable users to allocate and de-allocate data in the device data environment without being restricted to lexically structured scope.

Although the specification has been available for a significant length of time, there has been limited compiler support, and as such the there are not yet many applications that are taking advantage of OpenMP 4.5.

A. Benchmarks

As part of this research it was necessary to prepare and port a number of kernels and applications that would be used in the performance analysis. It was important that each of the benchmarks could offer some distinct insight into any potential performance differences, and covered a wide-ranging set of possible algorithmic patterns.

- **Linear Algebra Kernels** - Some of the kernels are simplistic, such as the *Vec Add* kernel, and represent the least challenge in terms of code generation. The matrix multiplication kernels have well-known optimizations for GPUs, but there is additional challenge in translating this to OpenMP in a performance portable manner [9].
- **Indirection Kernels** The indirection kernels are simple but can present a unique challenge for parallelization, and do not necessarily perform well [6].
- **Stencil Kernels** - We use simple stencils from 2pt to 27pt in 2D and 3D, as they represent algorithms that express some locality and benefit from the use of halo regions.
- **Wavefront Parallelism** - Wavefront parallelism results in inconsistent amounts of work being queued on a device, and requires multiple small kernel invocations.
- **Compute Bound Kernels** - All of the kernels presented up until this point are memory bandwidth bound, and so we create an artificial compute-bound kernel that executes 128 FMAs.

The mini-apps focus upon specific scientific problem domains, and will expose insights into the performance issues that might arise with full-scale scientific applications.

- **TeaLeaf** - A mature heat diffusion mini-app that solves the heat conduction equation on a structured grid using an implicit matrix-free solve. We first look at a performance-critical kernel present in the Chebyshev solver and later extend our research to investigate the performance of the whole application [4].
- **CloverLeaf** - A mature hydrodynamics mini-app, that solves Euler's equations explicitly across a structured mesh. As with TeaLeaf, we initially experiment with a single kernel and then extend our analysis to the entire application [10].
- **SNAP** - An adaptation of the mature discrete ordinates particle transport application that has been restructured to take advantage of GPUs. We consider the performance critical kernel of this application in isolation, excluding checks for time-dependence and negative flux fix-up, and then continue to fully investigate the performance of the application as a whole [11].

We expect that by using this diverse set of benchmarks we can deliver performance optimizations that will have an effect on many problem domains.

B. Hardware

All of the performance data was collected on an NVIDIA K40m GPU resident on a POWER8 machine, but we also verified our results on a K40c GPU hosted elsewhere. The K40m can achieve around 200 GB/s sustained memory bandwidth, with ECC turned on, and 1.4 TFlops/s in 64-bit. To compile the CUDA code we used CUDA version 7.5, and the OpenMP 4.5 code was compiled with an early closed-source version of the Clang compiler fork that is now available as an open-source branch[1].

C. Porting

Each of the kernels and applications was ported and tuned such that it performed as well as we could achieve with OpenMP 4.5 and CUDA. The only caveat to this is that we make our initial comparisons against CUDA kernels that do not exploit shared memory. We later show that there is significant optimization potential provided by programming to target shared memory, and propose an extension to OpenMP 4.5 for this purpose (Section VIII). In some cases we started with code already optimized for an earlier fork of the Clang compiler that implemented OpenMP 4.5, whilst for others we performed the porting from scratch.

The porting experience was straightforward, as the compiler is now feature-complete. As the benchmarks and applications are already optimized for parallel execution, the combined construct **target teams distribute parallel for** was suitable

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

for the majority of the kernels. Using this construct guarantees that a particular loop nest is tight and will not contain serial regions. As such, the code generation path does not have to consider more complicated cases of data sharing and parallel execution, making it easier to generate clean code [12]. In those cases where we were not able to use the full combined construct, we discuss the performance consequences.

IV. Performance Analysis

Our performance analysis progresses through multiple stages, each focusing on comparing the OpenMP 4.5 performance results and those achieved by an optimal CUDA implementation of each of the kernels. Our main intention is to uncover specific reasons for performance differences between the two and, where possible, design and implement some solution.

A. Approach

We start with an initial assessment of the performance of each of our simple micro-benchmark kernels. In order to create a fair load for the GPU, we use a problem size of 4096x4096 or 256x256x256 cells in double precision, which means each array is 128MB in size. Having collected the results and observed the most significant performance issues, we engage in manual analysis of the generated PTX outputs, comparing them to the *best case*: the PTX generated by *nvcc* for our CUDA benchmarks.

Our observations then inform manipulation of the generated intermediate PTX outputs for OpenMP 4.5, in order to measure the performance again and observe which changes optimize performance. Where necessary, we have made adjustments to the compiler, and make suggestions for how robust solutions could be implemented.

Taking the lessons learnt from the benchmarks, we use the same approach for the mini-apps, uncovering a range of additional issues that are introduced when you package multiple kernels into a single application. For the mini-apps we also consider the wallclock results as they are reported by the application, in order to make sure there are no intra-kernel overheads introduced with the implementation.

B. Initial Performance

The results shown in Figure 1 demonstrate that there is a significant scope for performance improvement, with kernels like the *27pt Stencil* and *Column Indirection* requiring 3x the runtime of CUDA. We also expected the first four vector operations to perform close to 100% of the performance of CUDA, as there should be little overhead in the generated code given their simplicity. This lead us to believe that there are either unnecessary overheads being added into each of the kernels, or the balance of teams and threads needed to be adjusted to increase the performance to expected levels.

Taking the *Vector Add* kernel as an example, we know that this kernel is strictly memory bandwidth bound, and so the overheads could not be attributable to additional instructions. It was possible that the performance degradation was being

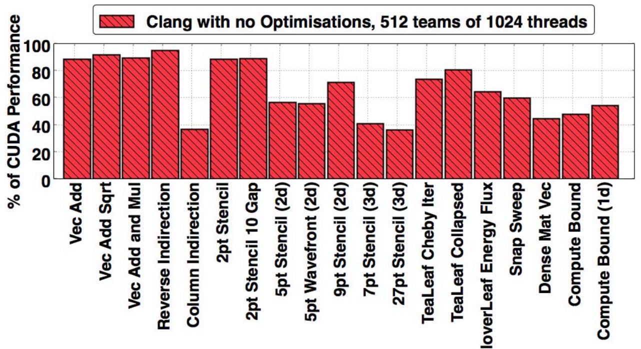

Fig. 1. Compares the performance of Clang OpenMP 4.5 with CUDA for the suite of kernels (higher is better).

TABLE I
Comparing the Clang OpenMP 4.5 and CUDA registers.

Benchmark	Clang *register*	CUDA *registers*	Diff (x)	Occupancy
Vec Add	20	10	2	95%
Vec Add Sqrt	28	18	1.6	93%
Vec Add and Mul	20	12	1.7	96%
Reverse Indirect	33	10	3.3	47%
Column Indirect	65	10	6.7	42%
2pt Stencil	18	10	1.8	94%
5pt Stencil (2d)	33	20	3.3	49%
5pt Wavefront	33	22	3.3	59%
9pt Stencil (2d)	33	23	1.4	49%
7pt Stencil (3d)	37	21	1.8	49%
27pt Stencil (3d)	36	35	1.0	49%
TeaLeaf Cheby	44	28	1.6	49%
CloverLeaf Flux	48	32	1.5	50%
SNAP Sweep	49	26	1.9	44%
Dense Mat Vec	28	34	0.8	13%
Compute Bound	12	8	1.5	98%

caused because of some inhibition of coalescing, but if that were the case the performance would have been more greatly affected than 15%.

The reason for the poor performance of the *Column Indirection* kernel is that the loops could not be collapsed using the **collapse** clause, because of the indirection that is present on the inner loop. As the inner loop could not be collapsed, the iteration space available in the outer loop was only 4096 in our test cases. This means that using the preferred combined construct **target teams distribute parallel for** would result in the routine starving the GPU of work.

As the stencil kernels grow in numbers of points, their relative performance progressively worsens, which we hypothesized might be caused by register pressure reducing their occupancy on the device. To investigate this theory we collected the register counts and occupancy values in order to understand whether this was the case (Table I).

Although the number of registers required by the Clang

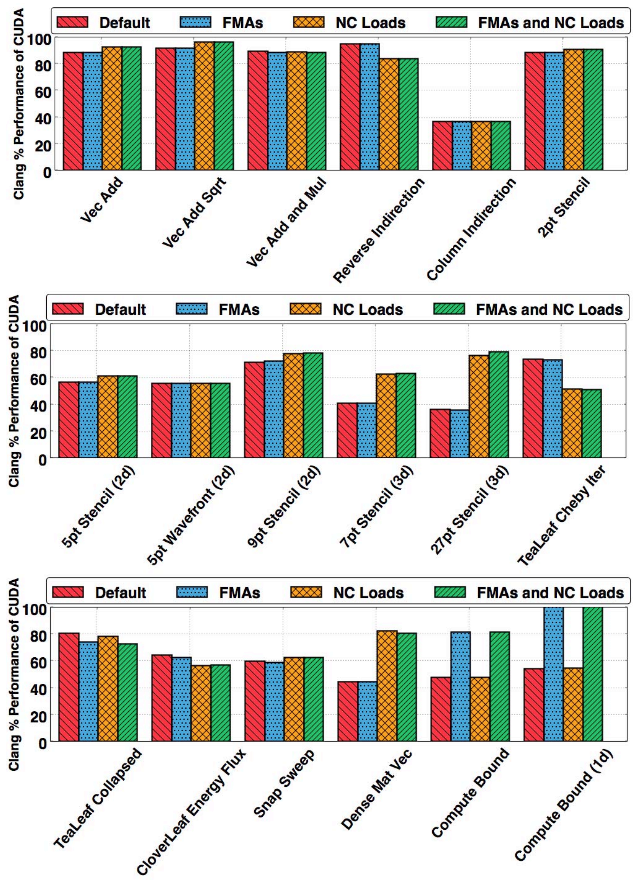

Fig. 2. Compares the performance of Clang OpenMP 4.5 with CUDA for the suite of kernels.

implementation is significantly higher, the occupancy of most of the kernels generated by OpenMP is acceptable. The only kernel that demonstrates a cause for concern in terms of occupancy is the *Dense Mat Vec* kernel. The dense matrix-vector multiply actually represents a work starved algorithm for our problem size, as the inner loop cannot be collapsed. Of course, this situation is identical for both of our implementations, and so we would not expect the performance to be so significantly divergent.

For the simplified applications kernels, we can see that the performance achieved is quite low compared to CUDA, although some performance improvement was observed by manually collapsing TeaLeaf, which will be discussed in Section IV-F. Finally, the compute-bound kernel demonstrates around a 50% reduction in performance compared to CUDA, which is discussed in Section IV-C.

C. *Fused-Multiply-Add*

Upon analyzing the PTX, it was clear that the OpenMP 4.5 compiler was not generating fused-multiply-add instructions, which we expected to be impacting performance to some extent. We had mainly compiled a set of memory-bandwidth bound codes, and had a single compute-bound benchmark, which runs 128 multiply-add expressions that can be transformed into FMAs. The initial results presented in Figure 1 demonstrate that the compute-bound algorithm achieves only 50% of peak performance achieved by CUDA, which makes sense as FMAs convert two instructions into one (Listing 1).

Listing 1. Fused-multiply-add instruction.

```
mul.rn.f64 %fd3, %fd2, %fd1;
add.rn.f64 %fd5, %fd4, %fd3;

transforms into

fma.rn.f64 %fd4, %fd2, %fd1, %fd3;
```

After some investigation we discovered that FMA instructions can be enabled via the compiler flag *-ffp-contract=fast*. Surprisingly the functionality is not enabled by default with the -O3 compiler flag. However, this flag conveniently provides the user with the ability to control the optimization on a per file basis, such that any kernels that do not benefit from it can be excluded from the optimization pass.

The results in Figure 2 show that the FMA instruction does not improve performance in any cases other than the *Compute Bound* kernels, which achieve a nearly 2x improvement in performance. In the case of the manually collapsed TeaLeaf implementation, the FMA instruction actually degrades the performance by more than 5%. This is surprising given that the kernel is strictly memory-bandwidth bound, but we have not yet been able to determine the cause.

D. *Non-Coherent Loads*

Another clear difference between the PTX outputs of CUDA and OpenMP 4.5 was the absence of non-coherent loads. There are two different paths through cache that data can take when being transferred from device memory, through conventional L2 cache, or through the non-coherent read-only cache, also known as texture cache. Data transferred via the non-coherent cache will benefit from higher memory bandwidth but experience higher latency [13]. In order to test this theory we intercepted the Clang compilation process, carefully assessing what data was only accessed only for reads, and changed them to pass through non-coherent cache (Listing 2).

Listing 2. The non-coherent load instruction.

```
ld.global.f64 %fd1, [%rd1];

transforms into

ld.global.nc.f64 %fd1, [%rd1];
```

We can see from the results in Figure 2 that performance is not improved in all cases by the inclusion of non-coherent loads, but in some cases the performance gain is significant. In particular, the 27pt stencil achieves around a 2x performance improvement, which is a consequence of the extensive locality present in the algorithm.

Our observation here is that non-coherent loads prove essential for achieving good performance with some kernels.

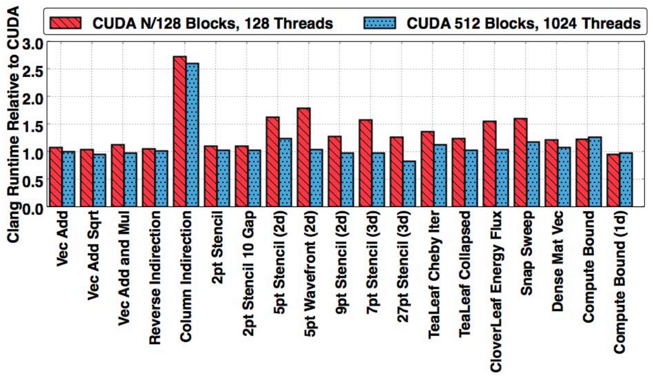

Fig. 3. The performance of the best Clang implementation compared to the original CUDA suite, and the suite that have an equivalent block and thread size to Clang.

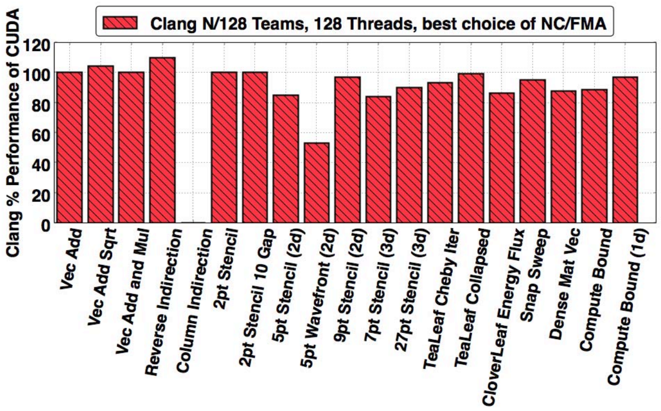

Fig. 4. The performance achieved by Clang compared to CUDA when using equivalent balance of teams and threads.

Therefore, we believe that it is imperative that some solution is implemented in the compiler to allow such loads to be generated, and discuss our approach in Section VI.

E. Team / Thread Balance

At this stage we recognized that the generated code matched the best case CUDA so closely that it might be useful to consider other factors. As such, we considered that the performance was likely being affected by the differences in thread partitioning between CUDA and Clang.

Clang's default behavior was to select 512 teams of 1024 threads, where the team limit was an upper bound that could not be increased with the **num_teams** directive. This upper bound was in place because there are book-keeping overheads required for data sharing when not using the combined construct **target teams distribute parallel for**. Essentially, some state was stored for each team, which imposed a limit on the number that could be active.

Unfortunately, the limitation with respect to the number of teams placed constraints on the flexibility of tuning the balance of teams and threads. It was not immediately clear how much effort would be involved in actually modifying the compiler to change this default behavior for the combined construct, and so we instead devised a simple check to convince us that the block sizes could be optimized beyond the default values. In our CUDA suite of benchmarks we created a new variant of each kernel that constrained the block size to 512, and the number of threads to 1024, increasing the amount of work that each of the threads performed.

We were hoping that by changing the performance of CUDA, we would observe that the performance difference between the two implementations would reduce, and that is exactly the result we present in Figure 3. In some cases the performance actually overshoots, and the Clang implementation performs better than the CUDA version. As we are primarily interested in comparing the best possible performance of both implementations, and this is a heuristic analysis that can introduce other issues, we do not consider the result as more than an indication.

Having completed this analysis, we knew that the performance difference between the two implementations was likely being affected by this issue and so we adjusted the limit within the OpenMP runtime and re-conducted our previous experiments to find the best performing kernels.

In Figure 4 the performance has started to approach the best performance achieved with CUDA in many cases. There are some important features of the results, in particular the case of *Vec Add Sqrt* and *Reverse Indirection*, which both achieve marginally improved performance compared to CUDA.

Generally we would consider results such as this to be quite dubious, however in this case we were able to uncover the precise reason for the performance disparity. The PTX assembly generated by the CUDA implementation includes non-coherent loads for both of those kernels, and we observe that our best results for those kernels are actually achieved with non-coherent loads omitted. As the algorithm is not latency-dependent, we do not expect that the increased latency has created this disparity, but we have not been able to understand exactly why this does occur.

The result of the 5pt wavefront appears to be one scenario where the performance does not improve with the changes to team size. We currently believe that this particular kernel needs more considered tuning of the team/thread balance and plan to do this in the future.

F. Collapse Implementation

You will note that the *Column Indirection* kernel does not have a result in Figure 4, and this is because we were not able to successfully collapse the inner loop. As a consequence, we were not able to use the combined construct and the optimised code generation scheme, which meant that the defaults for teams and threads could not be adjusted for the kernel. The reason that the kernel could not be collapsed is that the inner loop bound resides within an array, and the compiler is not currently capable of handling this form of collapse, although it has been noted as a feature for future implementation.

In some situations we observed that there was a significant difference between kernels that utilized the standard collapse implementation and underwent manual collapsing. As an example, we have included two versions of the TeaLeaf kernel in Figure 1, *TeaLeaf Cheby Iter* and *TeaLeaf Collapsed*, where the latter has been manually collapsed, and achieves noticeably better performance.

We initially observed that collapsed loops introduce two code blocks before the loop body is branched into. This code handles the re-forming of indices, checking for overflow and performing 64 bit divisions and remainders if an overflow occurred, or faster 32 bit versions if not. We note that this would likely have little effect on our memory bandwidth bound codes, and the 32 bit path will be followed in all cases as our problem size will not lead to an overflow.

Aside from this, the generated code contained little divergence from the code generated by CUDA. Given that the set of benchmarks exhibiting improved performance for manual collapse included memory bandwidth bound implementations, the branching overhead is unlikely to be the cause of the performance degradation.

V. Mini-App Performance Analysis

Having observed a number of performance effects that occur when running the simplified benchmarks, we extended this work to investigate the performance of the larger mini-apps. This extension provides another perspective on the performance issues in the compiler, and offers more concrete insights that will be relevant to optimizing for the real applications.

Where results of kernels are presented, the numbers have been collected with *nvprof* unless otherwise mentioned. Where *wallclock* values are given, the values are collected by timing mechanisms in the particular application.

A. TeaLeaf

The TeaLeaf mini-app was originally developed as part of the UK Mini Apps Consortium (UKMAC) [14], and belongs to the Mantevo project, which contains a set of proxy applications [15]. The application contains multiple linear solvers for the heat conduction equation, each solving the problem implicitly using matrix-free methods. Of those linear solvers we investigate the performance of the conjugate gradient (CG) solver.

The porting exercise was straightforward as we started from a version already optimized for the earlier Clang-YKT branch. It was essential to collapse all of the loops throughout the application, and all of the critical loops were parallelized using the combined construct **target teams distribute parallel for**.

In our performance experiment we ran a single iteration of the 4096x4096 problem size, as used in our prior research because it represents the point of mesh convergence [4]. The results are shown in Table II, with the four kernels shown accounting for the major share of performance.

1) Team / thread balance: Following on from our previous observations, we adjusted the application so that each kernel would launch N/128 teams containing 128 threads, where N is the trip count of the loop. Unfortunately, this meant that our application did not successfully pass validation. We isolated

TABLE II
Initial performance results for the TeaLeaf kernels, using OpenMP 4.5 and CUDA.

		Clang			
Kernel	**CUDA**	**Default**	**Teams**	**NC Loads**	**Profiler**
Calc P	5.7s	8.7s	6.1s	5.8s	6.2s
Calc W	7.7s	13.2s	11.7s	10.1s	13.0s
Calc UR	11.8s	15.6s	14.8s	13.8s	16.7s
Wallclock	26.1s	46.0s	41.7s	38.5s	38.5s

the cause of this issue to be when kernels with reductions scheduled too many teams, and reduced the number of teams below this limit.

2) Potential offloading overheads: We did observe that there was a significant disparity between the execution time of the kernel collected by our function-level profiling, and the time shown by *nvprof*. Looking at the results presented in Table II: **NC Loads**, we can see that the total profile of the kernels is 29.7s, but the wallclock is 38.5s, leaving 23% of the runtime unaccounted for, and only negligible time spent in other kernels. It can particularly be noted that the *Calc UR* kernel ran for 13.8s according to *nvprof* for the best performing implementation, but 16.7s according to timing taken before and after the kernel.

Given that the most significant disparity is seen with the kernels *Calc W* and *Calc UR*, it is likely that the reduction implementation is introducing an additional overhead that is not being captured by the profiling. There is currently work being undertaken to improve the performance of the reduction clause in general, and we expect that once complete it will have a significant influence on the performance of this application.

We expected at this stage that there might be issues with the register allocation, but were surprised to see that the occupancy of each of the kernels was over 70% for all of the key kernels. Given that the best performance we achieve is a 48% penalty compared to CUDA, we expect that there are some significant optimizations outstanding, and we will continue to investigate the application in the future.

B. CloverLeaf

The CloverLeaf mini-app was also originally created as part of UKMAC and belongs to the Mantevo project. The application hosts a hydrodynamics solver that progresses through a series of kernels that solve the Euler equations explicitly across a spatially decomposed grid conserving momentum, mass and energy. Extensive performance analysis has been conducted on the application and we expect the CUDA results to be the best possible results to be achievable for this particular problem.

The porting process was simplified even further than our previous efforts with the use of the **target enter data** clause, which meant that we could map the data immediately following the allocation routine, and then offload all kernels

TABLE III
BEST ACHIEVED PERFORMANCE DIFFERENCE BETWEEN CLOVERLEAF CUDA AND OPENMP 4.5.

		Clang			
Kernel	**CUDA**	**Initial**	**Teams**	**NC**	**Registers**
Wallclock	11.5s	15.7s	14.1s	13.6s	12.8s

independently [6]. All of the kernels use the combined construct **target teams distribute parallel for**, and we found the best performance was achieved by collapsing the loops of all kernels.

By optimizing the thread / team balance in the same manner as TeaLeaf, the application successfully runs with a 22% performance penalty compared to CUDA. Extending this to manually inject non-coherent loads improved the performance marginally, reducing the difference to 19% (Table III: **NC**).

C. Register Allocation

Investigating the reasons for this 19% performance difference, it is noticeable that some of the kernels are suffering from register allocation issues. The *viscosity* kernel, for instance, in CUDA has 0.6 occupancy, while the OpenMP 4.5 code has 0.3 occupancy. This correlates with CUDA using 42 registers, and OpenMP using 85, and the kernel suffers from a significant performance penalty, running in 1.18s with OpenMP and 0.66s with CUDA.

It was possible to improve this performance difference significantly by slightly adapting the algorithm so that indices were more transparent to the compiler. This meant that the relative address offsets that are used to access some of the memory locations could be optimized, avoiding calculating them at runtime and decreasing the amount of registers required from 85 to 72. We further passed the *maxrregcount* flag to PTXAS to limit the number of registers to 64, and saw even better performance, of around 11% penalty compared to CUDA (Table III: **Registers**).

This result is impressive, demonstrating that good wallclock performance is achievable with the compiler. We believe that, with the improvements we propose in Section VII, this gap can be closed even further.

D. SNAP

The SNAP application contains two key routines: a sweep kernel that uses wavefront parallelism, and a pair of reduction kernels. We were able to port the full mini-app to use the Clang OpenMP 4.5 implementation, with the dataset resident on the device for the duration of the solve.

The test case chosen in this investigation is the most realistic possible with this mini-app, closely matching the problem parameters that might be used in a real application. In particular, the spatial domain is comprised of a single pencil of data that has dimensions $4 \times 4 \times 400$ (x, y, z). We use this shape because of the MPI decomposition scheme used by the application, which means each node will work on a single pencil-shaped domain [11].

TABLE IV
BEST ACHIEVED PERFORMANCE DIFFERENCE BETWEEN SNAP CUDA AND OPENMP 4.5.

		Clang			
Kernel	**CUDA**	**Default**	**Teams**	**Smem/NC**	**Reg**
Sweep	1.48s	3.74s	3.09s	2.94s	2.04s
Reduce Flux	0.41s	3.15s	3.17s	0.39s	0.36s
Reduce Mom	1.07s	4.11s	4.19s	1.27s	1.27s
Wallclock	4.26s	13.50s	12.96s	7.12s	6.03s

Note: **Smem/NC** = shared memory and non coherent load optimization, and **Reg** = register optimization.

The consequence of this orientation is that the wavefronts appear to propagate through the space along the z-dimension, leading to planes that contains $x \times y$ cells. We also use a typical setting for the angles and groups of 32, and 136 respectively.

1) Reduction Kernel: The first challenge that we encountered was attempting to express the same reduction operation as is performed in the optimized reference CUDA implementation. Initially, we had to opt for a sub-optimal solution, which performed an order of magnitude slower than the CUDA implementation. Without a direct interface through OpenMP 4.5 to shared memory, it is not immediately apparent how such a scheme can be implemented. We discuss in section VIII our approach to accessing shared memory with the Clang compiler, and include results for the improved SNAP reduction.

2) Sweep Kernel: Given that we could not express the reduction scheme in the preferred manner with simple directives, we focused our early efforts upon optimizing the sweep kernel. Our preliminary results (Table IV: **Default**) show that CUDA performs the sweep kernel 2.6x faster than OpenMP 4.5, which is significantly worse performance than we expected. This showed that some more issues had surfaced as the problem dimensions had changed and the kernel was embedded inside a more typical application flow.

Upon inspection of the PTX assembly, it was apparent that there were many opportunities in the sweep kernel for FMA instructions, but we knew the kernel was memory bound (latency and bandwidth) and saw no performance improvement. Adjusting the number of teams and threads made less difference for the sweep kernels than we have seen for other kernels in this investigation. The reason for the disparity is that the kernel exposes little parallelism at the loop level. Our optimization could at best alter the scheme of 512 teams of 384 threads to 544 teams of 128 threads (for the majority of calls), improving performance by around 20% (Table IV: **Teams**).

The sweep kernel written in CUDA was utilizing 58 registers, while the kernel written in OpenMP 4.5 was using 150 registers. The reason for the greatly inflated register count compared to our micro-benchmark is that our simplified kernel did not include any branches for time dependence or negative flux fix-up, which require a number of additional registers.

We collected the average achieved occupancy, which was 0.18 for the OpenMP 4.5 version, and 0.43 for CUDA, which shows a 2.4x better utilization by the CUDA kernels. This correlates to the performance difference observed, and so we strongly believed at this stage that register pressure was the principle cause of our performance degradation.

3) Register Tuning: We were able to utilize the *maxrregcount* flag to improve performance as with CloverLeaf. Through tuning the number of registers we were able to observe improved performance, and brought the kernel to within 50% of the CUDA performance by limiting the number of registers to 82 (Table IV: **Reg**). Of course, this will be improved even further once changes are made to the compiler to tackle the register allocation issues, discussed in Section VII.

VI. Non-Coherent Loads

Following on from the performance analysis in Section VI, we could see it is important that non-coherent loads are enabled within the compiler, but had two options for actually implementing this.

- **Automatically locate data that is accessed in a read-only manner and generate NC loads**
  This approach is the least effort for the developer, and matches the behavior that we have observed in the CUDA platform.
- **Allow the developer to hint that memory is readonly**
  This approach imposes some effort on the developer, and might be onerous in some applications, but means that developers can avoid the potential penalty associated with non-coherent loads when necessary.

Automatic analysis can leverage the functionality present in the CUDA platform to generate non-coherent loads, but custom analysis will likely come up against challenges such as when kernels contain functions that reside in different compilation units. We believe that allowing the developer to hint that a particular variable is read-only through some declaration of const-ness better suits the nature of OpenMP. In practice, we recognize that some hybrid implementation that can support both approaches might be the best approach.

Listing 3. OpenMP 5 hint for const array section.

```
#pragma omp target map(const, to: a[:N])
{
  ...
```

In order to give the user control over the generation of non-coherent loads we propose an extension to the OpenMP specification to support a read-only hint. The extension, shown in Listing 3, is a **map-type-modifier** that hints to the compiler that the array section being copied by a map clause is to be treated as constant for the kernel. Assuming that the pointer passes alias checks, the data would then be passed through non-coherent cache.

This would allow the developer to mark arrays with high locality within a kernel, whilst ignoring those that wouldn't benefit from non-coherent cache. Further to this, giving the option as a hint still allows a cost model to be developed which can decide when the non-coherent loads should be applied.

In order to provide users of the existing Clang OpenMP 4.5 implementation with the ability to enable non-coherent loads immediately, we adjust the compiler directly. We added a new compiler flag *-fopenmp-nonaliased-maps*, which ensures all pointers mapped within a target regions will be marked with the *restrict* qualifier, enabling the NVPTX backend to generate non-coherent loads. We plan to later extend this work and attempt to develop a cost model that can improve performance by only using non-coherent loads when sufficient locality is expressed.

VII. Register Allocation

We were able to uncover a significant issue with the register allocation, that was causing a majority of the register inflation compared to CUDA. Increasing the amount of work per kernel and reducing the number of teams meant that an additional loop had to be added to each of the kernels, which we now show affects the register allocation.

Listing 4. The work loop causing register allocation issues.

```
LBB0_01:
  ...
  mul.rn.f64      %fd1, %fd2, %fd3;
  st.global.f64   [%rd1], %fd1;
LBB0_02:
  add.s64         %rd2, %rd2, %rd3;
  setp.lt.s64     %p1, %rd2, %rd4;
  @%p1 bra        LBB0_1;
LBB0_03:
  ret;
```

Listing 4 is a simplified example to demonstrate the issue. From the start of the body, LBB0_01, to the end, LBB0_02, a number of registers will be allocated for calculations, many of which will only have a lifetime of a finite number of lines. The tail of the loop, LBB0_02, will ensure those threads with multiple iterations can re-compute the body again until they have completed all iterations.

This has the unfortunate consequence that register allocation is much harder to reason about. PTXAS appears to believe that the registers within the loop body might need to be used again and so keeps them alive for the entire loop body. If instead the branch LBB0_02 was not present, then the PTXAS assembler could see that many of the registers can be re-used, reducing the net requirement.

We have been able to prove that removing this loop from the PTX greatly improves the register allocation, bringing it much closer to our CUDA codes. For instance, the SNAP sweep kernel that required 150 registers improved to 88 registers with this fix. Implementing this change in the compiler will be dependent upon the kernels being launched with N threads and M/N teams, where M is the length of the iteration space. While we cannot easily make this adjustment to the compiler, we are going to actively research this issue going forward.

VIII. Shared Memory

Our research so far demonstrated that the Clang implementation of OpenMP 4.5 can achieve good performance relative to CUDA, however there are some optimizations that we have purposefully excluded from the CUDA kernels. In particular, we have not used kernels that benefit directly from optimizations where memory is blocked into the shared memory space. Of course, this greatly reduces the potential for optimization with some algorithms, the most canonical example being the matrix-multiply.

A. Matrix Multiplication

In order to make some initial assessment of the performance of the shared capabilities exposed by the Clang OpenMP 4.5 implementation, we investigate some variations of the dense matrix-matrix multiplication algorithm. We chose a large test case of multiplying two square matrices of size 8096x8096, and storing the result in a third matrix.

We used two variants in CUDA: (1) a naive approach that used each thread to perform the entire multiplication for a single solution element in global memory, and (2) an implementation adapted from the CUDA programming guide which blocks the computations into 16x16 blocks in shared memory. The blocking kernel exploits the potential for locality within a block, allowing significantly more re-use than with the original version.

Our performance analysis showed that the naive CUDA kernel took 28.4s for our test problem, while the shared kernel took only 3.1s. This 9x performance improvement demonstrates that it is important to the future of OpenMP that this optimization is considered, in time for developers to begin using OpenMP in real applications and libraries targeting NVIDIA GPUs.

We wanted to demonstrate that a shared memory option exposed via OpenMP could be straightforward, and developing some form of *scratchpad* support can benefit developers who are using OpenMP in real application. In order to achieve this, we have taken advantage of a feature of Clang that allows you target specific address spaces. The **address_space(3)** attribute was used to block the sub-matrices into memory shared by each team.

Listing 5. Accessing shared memory space.
```
#pragma omp target teams distribute
for(int ii = 0; ii < n; ++ii) {
  ...
  static __attribute((address_space(3)))
    double Ashared[BLOCK_SIZE];
}
```

Synchronization was an additional obstacle to implementing this successfully with OpenMP, as you are not able to synchronize within a **parallel for** because of the potential for threads not encountering the **barrier** if the iterations don't partition perfectly. This meant that we were not strictly able to utilize the improved code generation scheme provided by the Clang compiler for the **target teams distribute parallel for** combined construct.

In order to observe what performance would have been possible if we could synchronize within the optimal code generation scheme, we manually added synchronizations into the intermediate PTX files and measured the performance. The naive kernel for OpenMP 4.5 ran in 32.2s, but adjusting the kernel to use non-coherent loads worsened the performance to 35.5s. The kernel optimized for shared memory initially ran in 6.1s, and improved to 5.1s by changing all loads to pass through non-coherent cache. This result demonstrates that a 6.3x performance improvement is possible, but the code generation scheme for the currently sub-optimal constructs will need to be improved to achieve this in practice.

B. SNAP

There were also shared memory optimizations that could be achieved with the SNAP application, which hosts two reduction routines that each reduce a large multidimensional array by two dimensions. For our earliest attempt with OpenMP 4.5 we were forced to perform the local reductions in global memory, but this can be greatly improved by performing the reductions within a team using shared memory.

As observed with the matrix multiply kernel, we were not able to use the **barrier** directive from within a **parallel for**, which limited our ability to perform the optimization. We again intercepted the PTX and manually added the barriers, and we were able to bring the performance to within 20% of the optimized CUDA performance for those reduction routines (Table IV: **Smem/NC**).

In order to express the kernels in OpenMP, you would not use the combined construct, but instead distribute the outer loops across the teams and have each team the same width as the trip count of the inner loop.

Listing 6. Replacing the inner loop with a team parallel region.
```
for(int ii = 0; ii < outer; ++ii) {
  for(int jj = 0; jj < inner; ++jj) {
    a[ii*inner + jj] = 1.0;
  }
}

becomes

#pragma omp target teams distribute \
  thread_limit(inner)
for(int ii = 0; ii < outer; ++ii) {
  #pragma omp parallel
  {
    const int jj = omp_get_thread_num();
    a[ii*inner + jj] = 1.0;
  }
}
```

This ensures that any barriers will be encountered by all threads within the team. Unfortunately, the implementation is more complicated in terms of code generation, in part due to the necessity to handle interspersed serial and parallel regions within the distributed loop [12]. As such, the performance achieved by this implementation is actually worse than if our sub-optimal global memory implementation was utilized.

We will investigate to see if there is any viable approach to optimizing this in the near future.

C. Potential Solution

Handling the multi-tiered memory hierarchies in HPC applications is an area of significant research and receiving a lot of attention within the community [16]. It is likely that this problem will be tackled as part of the OpenMP 5.0 specification, but we believe that it can be handled simply when considering shared caches.

Listing 7. Declaring a scratchpad within an OpenMP kernel.

```
#pragma omp target teams distribute \
  thread_limit(inner)
for(int ii = 0; ii < outer; ++ii) {
  #pragma omp parallel
  {
    #pragma omp scratchpad
    double Ashared[BLOCK_SIZE];
    ...
  }
}
```

The approach proposed in Listing 7 is similar to how you would access shared memory with CUDA, and programming shared caches in other models. The directive is quite general, and could be extended to handle multiple levels of cache, and accept attributes based on the way an array is used. However, for its primary purpose, targeting programmable cache, we believe it is sufficient and readily implementable.

IX. Future Work

We noted a number of issues throughout this paper that we will continue to investigate. Some of the kernels did not achieve the same level of performance as CUDA, and we think it will be important to understand why. The **collapse** implementation of OpenMP 4.5 in Clang needs to be improved to handle indirection arrays.

Once the **reduction** clause has been optimized further, it will be useful to re-investigate the performance achieved by TeaLeaf. Finally, we believe it is essential that shared memory optimizations are enabled in the Clang OpenMP implementation and the specification, and we plan to investigate potential improvements to the code generation scheme for this purpose.

X. Related Work

There has been significant research over the last few years into the viability of using OpenMP to target accelerators. Liao et al. performed an early study targeting NVIDIA GPUs with their prototype implementation in the ROSE compiler [17]. Ozen et al. investigated the OpenMP 4.0 functionality implemented in the OmpSs compiler [18].

A precursory stage of work saw the development of a prototype implementation of OpenMP 4.0 in Clang, and significant research was conducted around the performance optimization of this implementation. Bercea et al. investigated the performance of an OpenMP 4.0 port of LULESH [19], while Bertolli et al. developed a novel approach to handling interspersed serial and parallel regions [12].

Extensive performance analysis has been conducted using the mini-apps TeaLeaf, CloverLeaf and SNAP. McIntosh-Smith et al. used CloverLeaf to investigate the performance portability of OpenCL [20], proving that it was possible with tuning of key parameters. Mallinson et al. also investigated the performance and portability of an OpenCL port of CloverLeaf. Herdman et al. analyzed the performance of CloverLeaf on NVIDIA GPUs using CUDA, OpenACC and OpenCL [2].

We have investigated the performance of a number of performance portable parallel programming models using TeaLeaf [4]. This work was extended to consider CloverLeaf, SNAP and BUDE with Cray's OpenMP 4.0 implementation [6]. More recently, we conducted an extensive performance analysis of compiler technologies implementing OpenMP 4.x on a range of target devices [21].

Hart et al. ported the CORAL benchmark code NekBone to target NVIDIA GPUs with Cray's OpenMP 4.0 compiler implementation [22]. Larrea et al. assessed the potential for OpenMP 4.0 to be performance portable, by analyzing performance on an NVIDIA GPU and Intel Xeon Phi [1].

Acknowledgments

We would like to extend our gratitude to the scientists at the U.S. DoE laboratories, and NVIDIA, for their support and guidance with optimizing the OpenMP compiler and runtime libraries for the OpenPOWER platform. This research is supported in part by the CORAL project LLNS Subcontract No. B604142.

XI. Conclusion

We were able to discover some differences between the code generated by Clang and CUDA, that manifested in varying performance differences for the kernels. Enabling FMAs was essential for our compute-bound tests, and could be simply enabled for an application using a compiler flag.

Non-coherent loads were essential in order to achieve reasonable performance on the GPU for kernels that exposed locality of reads, but we observed some cases where non-coherent loads could reduce the performance. As a consequence, we believe that users should be responsible for hinting when those loads can be generated for a particular kernel, allowing fine-grained performance tuning where required.

We found that restrictions on the number of teams present in the compiler had a drastic impact on performance, and considered adjustments to the compiler to support the selection of improved defaults. We also discovered the cause of significant register allocation issues currently experienced by the Clang compiler, and suggest future work to improve this.

Finally, we demonstrated that shared memory optimizations can be particularly important to performance, and that both the compiler implementation and specification need to be adjusted to support this. Having now discovered the majority of issues affecting performance in the current Clang OpenMP 4.5 implementation, we have either shown that good performance is possible now, or suggested how performance can be improved in the near future.

References

[1] V. G. V. Larrea, W. Joubert, M. G. Lopez, and O. Hernandez, “Early Experiences Writing Performance Portable OpenMP 4 Codes,” Cray Users Group, London, 2016.

[2] J. Herdman, W. Gaudin, S. McIntosh-Smith, M. Boulton, D. Beckingsale, A. Mallinson, and S. Jarvis, “Accelerating Hydrocodes with OpenACC, OpenCL and CUDA,” in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 465–471.

[3] 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.

[4] M. Martineau, S. McIntosh-Smith, M. Boulton, and W. Gaudin, “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*, ser. PMAM’16, 2016.

[5] H. C. Edwards, C. Trott, and D. Sunderland, “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[6] M. Martineau, S. McIntosh-Smith, and W. Gaudin, “Evaluating OpenMP 4.0’s Effectiveness as a Heterogeneous Parallel Programming Model,” in *Submitted to 21st International Workship on High-Level Parallel Programming Models and Supportive Environments*, ser. HIPS’16, 2016.

[7] C. Bertolli, S. Antao, G.-T. 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*, ser. LLVM ’15, 2015.

[8] OpenMP Architecture Review Board, “OpenMP Application Program Interface Version 4.0,” 2013.

[9] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun, “Fast implementation of DGEMM on Fermi GPU,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 35.

[10] A. Mallinson, D. Beckingsale, W. Gaudin *et al.*, “Towards Portable Performance for Explicit Hydrodynamics Codes,” 2013.

[11] T. Deakin, S. McIntosh-Smith, and W. Gaudin, *Many-Core Acceleration of a Discrete Ordinates Transport Mini-App at Extreme Scale*. Springer International Publishing, 2016, pp. 429–448.

[12] C. Bertolli, S. F. Antao, A. Eichenberger *et al.*, “Coordinating GPU threads for OpenMP 4.0 in LLVM,” in *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*. IEEE Press, 2014, pp. 12–21.

[13] N. Corporation, “CUDA C Programming Guide Version 7.5,” 2015.

[14] UKMAC, “UK Mini-App Consortium,” http://uk-mac.github.io, 2016.

[15] M. Heroux, D. Doerfler *et al.*, “Improving Performance via Mini-applications,” Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[16] I. Karlin, “Quad-Lab Proposal for Fundamental Cross Architecture Multi-Level Memory Support,” Presentation at DOE Centers of Execellence Performance Portability Meeting. Available from: https://asc.llnl.gov/DOE-COE-Mtg-2016/talks/2-05_Karlin.pdf, 2016.

[17] C. Liao, Y. Yan, B. de Supinski *et al.*, “Early Experiences with the OMP Accelerator Model,” in *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, 2013.

[18] 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.

[19] G. Bercea, C. Bertolli, S. Antao, A. Jacob *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.

[20] S. McIntosh-Smith, M. Boulton, D. Curran, and J. Price, “On the Performance Portability of Structured Grid Codes on Many-Core Computer Architectures,” in *Supercomputing*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8488, pp. 53–75.

[21] M. Martineau, J. Price, McIntosh-Smith, and W. Gaudin, “Pragmatic Performance Portability with OpenMP 4.x,” in *Proceedings of the International Workshop on OpenMP*. Springer, 2016.

[22] 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.