CCAMP: OpenMP and OpenACC Interoperable Framework

Jacob Lambert[1(✉)], Seyong Lee[2], Allen Malony[1], and Jeffrey S. Vetter[2]

[1] University of Oregon, Eugene, USA
{jlambert,malony}@cs.uoregon.ed
[2] Oak Ridge National Laboratory, Oak Ridge, USA
{lees2,vetter}@ornl.gov

Abstract. Heterogeneous systems have become a staple of the HPC environment. Several directive-based solutions, such as OpenMP and OpenACC, have been developed to alleviate the challenges of programming heterogeneous systems, and these standards strive to provide a single portable programming solution across heterogeneous environments. However, in many ways this goal has yet to be realized due to device-specific implementations and different levels of language support across compilers. In this framework we aim to analyze and address the different levels of optimization and compatibility between OpenACC and OpenMP programs and device compilers. We introduce the **CCAMP** framework, built on the OpenARC compiler, which implements language translation between Open**ACC** and Open**MP**, with the goal of exploiting the maturity of different device-specific compilers to maximize performance for a given architecture. We show that CCAMP allows us to generate code for a specific device-compiler combination given a device-agnostic OpenMP or OpenACC program, allowing compilation and execution of programs with specific directives on otherwise incompatible devices. CCAMP also provides a starting point for a more advanced interoperable framework that can effectively provide directive translation and device, compiler, and application specific optimizations.

Keywords: OpenMP · OpenACC · Directive-based programming · Heterogeneous computing · CCAMP

1 Introduction

Coincident with the breakdown of Dennard Scaling and the slowing of Moore's law, heterogeneous programming has emerged as an alternative to traditional homogeneous computation [11]. The explosion in popularity of GPGPU programming, and now other devices like many-core processors and FPGAs, has led to the development of several new low-level programming approaches in order to map computations to these specific devices. Low-level heterogeneous programming approaches like CUDA and OpenCL grant knowledgeable programmers the

© Springer Nature Switzerland AG 2020
U. Schwardmann et al. (Eds.): Euro-Par 2019 Workshops, LNCS 11997, pp. 357–369, 2020.
https://doi.org/10.1007/978-3-030-48340-1_28

ability to write applications catered specifically to unique devices in an attempt to maximize performance.

However, these low-level device-specific programming approaches sacrifice the functional and performance portability enjoyed by more traditional homogeneous implementations. Rewriting and maintaining different versions of the same applications for different devices can be unsustainable and error-prone. Furthermore, the low-level device-specific approaches are intimidating and inaccessible to less experienced programmers.

Several higher-level, directive-based, device-agnostic programming standards have emerged to address the issues with device-specific implementations. These standards aim to enable programmers to annotate a general, sequential application with simple instructions for parallelism, transferring much of the low-level specifics to the compiler. However, as we discuss in Sect. 2, these directive-based approaches come with their own set of issues as well, and the ideal performance and portability proposed by the standards do not match the current reality.

CCAMP, an OpenACC and OpenMP interoperability framework, exists to bridge the gap between the current realities of performance and portability within existing standard implementations, and initial goals and intentions of these directive-based standards. CCAMP also provides programmers who only have experience with one directive-based programming model an easy alternative to learning another model by providing a translation framework.

2 Background

A primary goal of the CCAMP framework is to allow programmers to fully utilize the OpenMP and OpenACC directive-based programming standards, which have become a popular option for high-level heterogeneous programming.

OpenMP [3] has been an essential tool in the general parallel programming environment for decades. With the introduction of directives in the 4.X+ standards, OpenMP has also become a viable tool for heterogeneous programming, offering a high-level, offload programming model alternative to languages like CUDA and OpenCL.

OpenACC [12] is a newer directive-based standard, originally developed as a high-level alternative to CUDA for GPU computing. While OpenACC differs from OpenMP with regard to high-level design principles, they share a common goal of providing programmers with a high-level approach to heterogeneous programming.

While both OpenMP 4.X+ and OpenACC directives provide a method for high-level heterogeneous programming, there exist several important issues and setbacks to using these standards.

A primary issue in the directive-based heterogeneous programming space is the lack of portability between programming models. Although the goal of both OpenMP and OpenACC is to provide a portable, high-performance, cross-platform solution, they are often at the mercy of vendor-specific compiler implementations. Many devices achieve high performance when using the vendor-compiler tied to that device, which often supports only one of OpenACC and

OpenMP. Typically, GPU-centric and CPU-centric ecosystems prefer OpenACC and OpenMP, respectively. However, even among compilers preferring a specific directive-based standard, the level of support and implementation strategies for the standard can vary greatly.

As a result of these issues, both OpenACC and OpenMP 4.X+ fail to achieve the goal of being portable solutions for heterogeneous systems. One way to address this gap would involve the development of an optimization that takes a device-agnostic input code in either OpenACC and OpenMP, and automatically generates device-optimized code specific to a target device and compiler combination. The CCAMP framework, with its baseline translation capabilities, is an initial effort to realize such an optimization framework.

The main contributions of this work are as follows:

- We introduce a novel baseline directive-translation framework, allowing programmers to automatically flow between standards to utilize the maturity of single-standard compilers on different devices (Sect. 3).
- We provide a commentary on the current status of the popular OpenACC and OpenMP compilers and their levels of support for the directive-based standards across an array of devices (Sect. 4).
- We evaluate the effectiveness of CCAMP's baseline translation using an array of different heterogeneous ecosystems. We demonstrate how our compiler-translated code can perform similarly or even better than hand-written code, and how CCAMP can allow programmers to execute translated code in ecosystems that may not support the original source language (Sect. 5).
- We discuss the future of CCAMP and the extensions needed to develop a fully-fledged framework capable of providing true interoperability between OpenACC and OpenMP (Sect. 6).

3 CCAMP Framework

In it's current state, the CCAMP framework consists of three baseline translations, built on top of the OpenARC [9] compiler framework:

- OpenMP 4.X+ to OpenACC
- OpenACC to OpenMP 4.X+
- OpenACC to OpenMP 3.0

As previous researchers have noted [1,13,14], many directives in OpenMP and OpenACC have a straightforward, one-to-one directive mapping. These include data movement, allocation, and update directives, entries to parallel regions, and general clauses like *if*, *collapse*, *reduction*. Similarly, many of the relevant API calls have analogous counterparts in both directive sets.

However, despite their surface-level similarities, fundamental differences in the core of the language designs lead to some challenges in the language translation process, especially when deciding how to map parallelism to a specified device.

3.1 OpenARC

CCAMP is built on top of the existing OpenARC [9] framework. OpenARC is a research-focused OpenACC compiler for heterogeneous systems, and already contains some language-translation features to generate device-specific code like OpenCL and CUDA, and OpenMP directive parsing capabilities, inherited from the base Cetus compiler infrastructure [4].

One of the primary strengths of OpenARC lies in its capabilities to allow quick prototyping of code transformations, which proved crucial when developing the transformations and optimizations for CCAMP. Essentially, CCAMP exists as a translation and optimization layer that follows OpenARC's initial lexical analysis and AST generation.

3.2 OpenMP 4.X+ to OpenACC Translation

By design, OpenMP is implemented as a prescriptive set of directives, explicitly specifying how parallelism in a program should be mapped to CPU threads or GPU cores. This prescriptive nature simplifies the OpenMP 4.X+ to OpenACC translation pass, as the burden of specifying parallelism is placed on the programmer instead of the compiler. Because of this, the prescriptive OpenMP parallelism clauses can be directly translated to descriptive OpenACC counterparts without additional compiler analysis.

However, there are still several OpenMP language constructs that don't allow for a direct translation or mapping, including OpenMP critical regions and tasking. These non-trivial translations require some additional compiler analysis for correct translation.

By design, the OpenACC standard does not contain a directive analogous to the OpenMP critical region. GPUs represented the main target architecture during the design of OpenACC, and synchronization constructs like critical regions typically lead to poor performance on GPUs. To prevent programmers from experiencing this pitfall, critical regions were intentionally omitted. However, one use of OpenMP critical regions can be efficiently mapped to GPUs: array-reductions.

Currently when encountering OpenMP critical regions in the OpenMP to OpenACC translation, CCAMP emits an error and terminates translation. However, CCAMP is designed to detect if an OpenMP critical region is used to encapsulate an array reduction, and can appropriately translate the reduction using OpenACC reduction clauses.

Another OpenMP construct that does not directly translate to OpenACC is the recently introduced task construct. OpenMP task translation is not currently supported by CCAMP, but will likely be a focus of future extensions.

3.3 OpenACC to OpenMP 4.X+ Translation

Unlike OpenMP, OpenACC is designed with a descriptive outlook. The core principle of OpenACC is that the directives allow a programmer to expose or

describe parallelism within a program, and shift the burden of mapping parallelism to hardware from the programmer to the compiler. OpenACC also contains prescriptive directives and clauses to allow the programmer explicitly specify the mapping of parallelism, but these directives are not mandatory.

This difference in fundamental design complicates the OpenACC to OpenMP 4.X+ translation, as we're required to generate a prescriptive output from a descriptive input. In CCAMP, we tackle this issue by applying a compiler analysis to automatically annotate ambiguous OpenACC directives with specific parallelism clauses. Using an optimizing-loop-directive-prepossessing pass, we can automatically assign OpenACC *gang* and *worker* clauses to un-annotated loops.

More specifically, CCAMP utilizes OpenARC's auto-parallelization pass to mark kernel inner loops as independent when possible, exploiting available parallelism. Marked loops are then annotated with OpenACC parallelization clauses before the direct substitution translations to OpenMP occur.

In addition to the differences in requirements for descriptive detail, CCAMP also addresses several low-level syntactical differences when translating OpenACC to OpenMP. For example, the requirements on the location of *reduction* clauses differ between the standards, and so CCAMP performs a reduction directive migration pass. Similarly, the requirements on the OpenMP *num_threads* and *simdlen* clauses require migration of the corresponding *num_workers* and *vector_length* OpenACC clauses during translation.

Interestingly, OpenMP lacks a clause analogous to the OpenACC *present* clause. To mimic the behavior of the OpenACC *present* clause, we use an *assert()* function call along with the OpenMP *omp_target_is_present()* API call.

3.4 OpenACC to OpenMP 3.0

Although OpenMP 4.X+ exists as a super-set of OpenMP 3.0-only directives, in some cases programmers may wish to restrict the translated output to only employ OpenMP 3.0 directives. On systems without offload capabilities, or without more modern compilers that support newer OpenMP directives and OpenACC, this translation pass allows execution of previously unsupported applications. Also, because OpenMP 3.0 directive sets are much older and more pervasive across compilers, even compilers that do support OpenMP 4.X+ directives may perform better using the older directives when targeting CPU devices.

CCAMP's OpenACC to OpenMP 3.0 translation pass is a straight-forward stripped-down alternative to the OpenACC to OpenMP 4.X+ pass. OpenACC parallel regions are mapped to OpenMP parallel regions, and outermost OpenACC loop parallelization clauses are mapped to OpenMP *parallel for* clauses. The innermost OpenACC parallelization clause is mapped to OpenMP *simd* clauses. Intermediate OpenACC parallelization clauses are ignored.

In general, this translation can be useful any time a programmer is targeting a CPU device with a compiler that may struggle with the OpenMP 4.X+ directives, which is far from rare. The converse of this translation, OpenMP 3.0 to OpenACC, is not currently included in CCAMP, as this would require

automatic generation of data movement directives, and more complicated analysis of multi-tier parallelism.

4 Experimental Setting

4.1 Benchmarks

We chose to evaluate the CCAMP framework using the SPEC Accel Benchmark Suite [7] for several reasons. Most importantly, SPEC Accel already contains hand-optimized OpenACC and OpenMP implementations of the same set of applications. This provided an ideal baseline against which to compare our code translated by CCAMP. Additionally, SPEC Accel is well-supported, well-documented, and representative of a wide array of common scientific programming applications. While SPEC Accel contains both C and Fortran applications, we only target the C applications, as CCAMP does not currently support Fortran OpenACC and OpenMP codes.

We used the following SPEC Accel applications in our evaluations:

- X03 ostencil, (303 for OpenACC, and 503 for OpenMP) a thermodynamics stencil kernel
- X14 omriq, an application widely used in the medical field
- X52 ep, an embarrassingly parallel application
- X54 cg, a conjugate gradient kernel
- X57 csp, a scalar penta-diagonal solver, and
- X70 bt, a block tri-diagonal solver for 3D PDEs

The X52, X54, X57, and X70 benchmarks are adapted from the NAS Parallel Benchmark Suite [2], a benchmark set widely used for evaluating performance on heterogeneous systems. We also initially explored evaluating CCAMP using the Rodinia benchmark suite. Like SPEC Accel, Rodinia contains both hand-optimized OpenACC and OpenMP implementations. However, the OpenMP offloading implementations in Rodinia are optimized specifically for Xeon Phi devices, and perform poorly on GPU devices. This shortcoming further motivates the necessity of a framework like CCAMP, which can be used to generate device-agnostic OpenMP code from the existing Rodinia OpenACC implementations.

4.2 Devices

We evaluated CCAMP using a wide array of the most commonly used CPU and GPU devices in heterogeneous programming. The different devices are each coupled with vendor-specific compilers, which typically exhibit a preference between OpenMP and OpenACC. This further motivates a fluid way to translate between directive sets.

We evaluated CCAMP using three CPU systems:

- Xeon CPU: Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10 GHz, 32 CPUs, 1 thread per core, 16 cores per socket, 2 sockets
- Xeon Phi: Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40 GHz, 272 CPUs, 4 threads per core, 68 cores per socket, 1 socket
- Power9: IBM POWER9, altivec supported, 176 CPUs, 4 threads per core, 22 cores per socket, 2 sockets

We also evaluated CCAMP using two GPU systems:

- P100: Nvidia Tesla P100-PCIE-12 GB (Pascal), Xeon CPU host (as mentioned above)
- V100: Nvidia Tesla V100 SXM2 16 GB (Volta), Power9 host (as mentioned above)[1]

4.3 Compilers

Across different devices, vendor-supplied compiler frameworks often achieve the best performance on a specific device. In the context of directive-based approaches, these vendor-supplied compilers may only support one of OpenACC and OpenMP, or may strongly prefer one over the other. One of the primary goals of CCAMP is to allow programmers to exploit this compatibility between devices and vendor-compilers regardless of the chosen directive-based approach (using language translation).

To evaluate the effectiveness of CCAMP, we employed a breadth of compilers, some tied to specific devices (IBM xlc, Intel icc) and others multi-platform (Clang clang, PGI pgcc).

IBM's *xlc* C/C++ compiler is restricted to the Power9 and attached V100 devices. Currently, this compiler only supports OpenMP, although it does support both OpenMP host and OpenMP offload computation models. We use IBM XL C/C++ for Linux, V16.1.1. For evaluations on the Power9 device, we use the flags "-O3 -qsmp=noauto:omp -qnooffload", and for the V100 device we use the flags "-O3 -qsmp=noauto:omp -qoffload".

Intel's *icc* C/C++ compiler currently only supports OpenMP, with support for both host Xeon CPU devices, and Xeon Phi devices through OpenMP offloading. For CCAMP evaluations on the Xeon Phi, we use icc version 19.0.1.144 (gcc version 4.8.5 compatibility), and the following flags: "-O3 -xMIC-AVX512 -qopenmp -qopenmp-offload=host".

The open-source LLVM-based C/C++ compiler *clang* is not tied to a specific device. While clang doesn't currently support OpenACC, it fully supports the OpenMP host computation model, and there are ongoing efforts to develop full support for the OpenMP offloading model. For evaluations using clang on the Xeon CPU, we use release version 8.0.0 (git tag llvmorg-8.0.0-rc5). Support for correct handling of math functions in clang's OpenMP offload model has only

[1] The Power9+V100 configuration is very similar that of the Summit supercomputer nodes.

recently been added. For this reason, we installed clang directly from the master branch (git hash 431dd94) for evaluations using clang and the P100 device. When targeting the Xeon CPU, we use the flags "-Ofast -fopenmp -fopenmp-targets=x86_64", and for the P100 device we use "-Ofast -fopenmp -fopenmp-targets=nvptx64".

Although PGI's *pgcc* C/C++ compiler is now tied to Nvidia, pgcc supports all of the devices used in this work. However, to limit the project scope we only evaluate CCAMP using pgcc on the Xeon CPU, P100, and V100 devices. The pgcc compiler is the only compiler used in the evaluations that currently supports OpenACC. PGI's pgcc supports OpenMP 3.0 and a subset of OpenMP 4.X+ directives, although they do not yet support data transfer directives, limiting the OpenMP evaluations to host CPU devices. On the Xeon CPU and P100 devices we use version 18.10-0 64-bit (Community Edition). On the V100 device, we use the slightly older 18.4 edition. On the Xeon CPU we use the flags "-V18.10 -fast -Mnouniform -acc -ta=multicore" for OpenACC programs, and "-V18.10 -fast -Mnouniform -mp -Mllvm" for OpenMP programs. On the P100 and V100 devices, which are only supported via OpenACC, we use the flags "-V18.10 -fast -Mfprelaxed -acc -ta=tesla:cc60" and "-V18.4 -fast -Mnouniform -acc -ta=tesla:cc70" respectively.

Ideally for a more fair evaluation we would have no variations in compiler versions across different devices. However, this became a challenge due to the different levels of access and privileges across ecosystems, and the goal of using the most recent compiler releases. In future extensions to CCAMP, we plan to rectify these inconsistencies.

While a complete list of the most commonly used compilers in heterogeneous programming would include the GNU C/C++ Compiler *gcc*, we chose to exclude it from this work in progress due to difficulties with installation for OpenMP and OpenACC offloading, and to limit the scope of the project. We fully intend to include gcc on future works extending CCAMP.

5 Evaluation Results

We evaluated the effectiveness of the CCAMP framework using an exhaustive approach, compiling and testing as many different applications with as many different device+compiler combinations as possible. This required a significant effort, including installing software across different devices, and wading through the different levels of support for the multiple compilers.

5.1 OpenACC to OpenMP 4.X+ Baseline Evaluation

To evaluate the effectiveness of CCAMP's OpenACC to OpenMP 4.X+ baseline translation pass, we first evaluated the hand-coded OpenMP 4.X+ applications in the SPEC Accel benchmark suite without applying any transformations or optimizations. We used the resulting execution times as a baseline by which to compare the execution times of our translated (from OpenACC) OpenMP 4.X+ code.

In Fig. 1, we see the results of this comparison. Each bar represents the average (across all benchmarks) ratio of the translated runtime divided by the hand-coded runtime. Values below 1 represent cases where the translated code performed better, while values above 1 represent cases where further improvements need to be made to the translation pass to match the hand-coded performance. While the translation pass still has room for improvement on some device+compiler combinations, it performs acceptably well for many of the other combinations.

Averaging results across different benchmarks certainly results in loss of information. However the very large number of experimental results across devices, compilers, and benchmarks required a heavy amount of aggregation for a simplistic overarching view of the relative performances between the original and translated codes. We aim to provide more detailed evaluations focused on specific applications in future works, including profiling analysis and investigations into differences in performance.

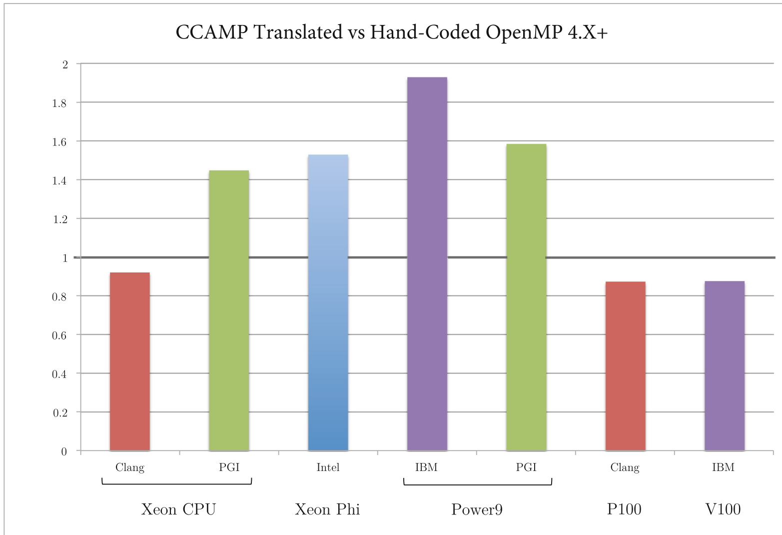

Fig. 1. Run-time comparison of translated OpenMP 4.X+ with hand-coded OpenMP 4.X+

5.2 OpenMP 4.X+ to OpenACC Baseline Evaluation

Similarly to the previous translation, to evaluate CCAMP's OpenMP 4.X+ to OpenACC translation pass we compare hand-coded OpenACC execution times with translated code times.

In Table 1, we list runtimes (in seconds) on the devices we used to evaluate this translation pass. Because the PGI compiler is the only compiler evaluated

that currently supports OpenACC (in this case translated from OpenMP 4.X+), PGI is used for all of the compilations in the figure.

We see that in most cases the translated code performs very similarly to the hand-coded counterparts. The dashed values represent cases where we failed to correctly execute the application, primarily due to unsupported features or errors pending correction in CCAMP.

Table 1. Run-time comparison of translated OpenACC with hand-coded OpenACC. Time in seconds.

Device	Translation	X03	X14	X52	X54	X57	X70
Xeon CPU	None	82.06	670.03	184.84	82.06	102.73	153.02
Xeon CPU	Baseline	81.39	670.15	184.58	182.17	88.18	-
P100	None	26.45	146.22	76.09	61.66	45.17	19.31
P100	Baseline	26.67	146.37	65.57	51.48	-	42.802
V100	None	12.33	38.84	47.71	33.04	20.19	9.14
V100	Baseline	13.95	33.35	52.91	31.42	-	25.03

6 Related Work

Several previous works explore the performance and portability of directive-based approaches across heterogeneous systems. In [8], Vergara et al. evaluate OpenMP applications on Power8 and Tesla devices using the IBM and clang compilers. In [10], Lopez et al. experiment with OpenACC and OpenMP implementations of core computational kernels, including Daxpy, Dgemv, Jacobi, and HACCmk. They evaluate the performance of these implementations using the Cray, Intel, and PGI compilers on Nvidia GPU and Intel Xeon Phi devices. In [6], Gayatri et al. implement a single material science kernel, and evaluate OpenMP 3.0, OpenMP 4.0, OpenACC, and CUDA implementations on Xeon CPUs, Xeon Phis, Nvidia P100s, and Nvidia V100s. This closely resembles the languages and devices evaluated in our work, although we evaluate multiple applications. Gayatri et al. also discuss their experiences with different compilers, including the PGI, Intel, IBM, and GCC compilers, and the then-current status of their directive-based language support. These works all highlight the high variability in performance of directive-based approaches across different compiler and device combinations, which helps to motivate the utility of a framework like CCAMP.

There are also several previous works that research the potential of an OpenACC and OpenMP translation framework. In [14], Wolfe explores this idea and discusses some obvious and some more-subtle challenges that would arise when implementing such a framework. He also discusses motivations and significance of developing such a framework, which are in line with the motivations we present

here. In [1], Sultana et al. present a prototype OpenACC to OpenMP translation scheme, which consists of a combination of automated directive translation performed using the Eclipse user interface and manual user-performed code restructuring. This work represents a promising first attempt to develop an automated translation framework, although they only evaluate a single benchmark and support only a subset of the OpenACC standard. In [13], Pino et al. describe a mapping between the most common directives of OpenACC and OpenMP, and compare the performance between the two different sets of directives on several SHOC and NAS benchmarks, but do not propose any automated scheme or framework to perform the actual translation. In [5], Denny et al. present an ongoing work to develop an OpenACC to OpenMP 4.5 translator (Clacc) within the clang compiler, as a means to allow clang to support OpenACC. Clacc represents a rigorous effort to develop a translation scheme supporting the full OpenACC standard, which accomplishes the goal of our OpenACC to OpenMP 4.5 baseline translation, but is constrained by the clang compiler, preventing it from utilizing the maturity of device-specific back-end compilers.

In contrast to previous works, that either represent only a conceptualization of a translation scheme, or in the case of Clacc [5] are tied to a specific device-level compilers, CCAMP presents an actual implementation of directive translation that is applicable across different device ecosystems and integrated with several different back-end compilers.

7 Conclusion

As systems become more exotic and specialized, the HPC community has experienced an increased demand for high-level, portable, programming solutions. While directive-based standards and approaches aim to provide a solution, they fail to realize this goal due to competition between vendor compilers, and inconsistent levels of standard support.

In this work, we present the CCAMP framework, with the goal of allowing programmers to seamlessly flow between different directive sets, enabling programmers to execute directive-based code on previously incompatible devices. We introduce two primary translation passes, and show that these passes can generate output code in a different directive context that performs similarly to hand-coded programs. We also provide a commentary on the current status of the different devices and compilers commonly used in heterogeneous programming.

In the future, we plan to develop and extend CCAMP in several ways. A primary goal is to develop an optimized translation pass that can generate not only generalized directive sets in different languages, but also directive sets specifically catered toward an indented target device. We also plan to incorporate other compilers (GCC, Clacc), and other devices (FPGAs). Finally we would like to expand our evaluations to include other benchmarks besides SPEC Accel.

Acknowledgements. This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

This manuscript has been co-authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan.

References

1. Arnold, G., Calvert, A., Overbey, J., Sultana, N.: From OpenACC to OpenMP 4: toward automatic translation. In: XCEDE 2016, Miami, FL (2016)
2. Bailey, D., Harris, T., Saphir, W., Van Der Wijngaart, R., Woo, A., Yarrow, M.: The NAS parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center (1995)
3. Dagum, L., Menon, R.: OpenMP: an industry-standard API for shared-memory programming. Comput. Sci. Eng. **1**, 46–55 (1998)
4. Dave, C., Bae, H., Min, S.J., Lee, S., Eigenmann, R., Midkiff, S.: Cetus: a source-to-source compiler infrastructure for multicores. IEEE Comput. **42**(12), 36–42 (2009). http://www.ecn.purdue.edu/ParaMount/publications/ieeecomputer-Cetus-09.pdf
5. Denny, J.E., Lee, S., Vetter, J.S.: CLACC: Translating OpenACC to OpenMP in clang. In: IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), pp. 18–29. IEEE (2018)
6. Gayatri, R., Yang, C., Kurth, T., Deslippe, J.: A case study for performance portability using OpenMP 4.5. In: Chandrasekaran, S., Juckeland, G., Wienke, S. (eds.) WACCPD 2018. LNCS, vol. 11381, pp. 75–95. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-12274-4_4
7. Juckeland, G., et al.: SPEC ACCEL: a standard application suite for measuring hardware accelerator performance. In: Jarvis, S.A., Wright, S.A., Hammond, S.D. (eds.) PMBS 2014. LNCS, vol. 8966, pp. 46–67. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17248-4_3
8. Larrea, V.V., Joubert, W., Lopez, M.G., Hernandez, O.: Early experiences writing performance portable OpenMP 4 codes. In: Proceedings of the Cray User Group Meeting, London, England (2016)
9. Lee, S., Vetter, J.: OpenARC: open accelerator research compiler for directive-based, efficient heterogeneous computing. In: Proceedings of the ACM Symposium on High-Performance Parallel and Distributed Computing, HPDC 2014, Short Paper, June 2014
10. Lopez, M.G., et al.: Towards achieving performance portability using directives for accelerators. In: 2016 Third Workshop on Accelerator Programming Using Directives (WACCPD), pp. 13–24. IEEE (2016)
11. Mittal, S., Vetter, J.S.: A survey of CPU-GPU heterogeneous computing techniques. ACM Comput. Surv. (CSUR) **47**(4), 69 (2015)
12. OpenACC: OpenACC: directives for accelerators (2011). http://www.openacc.org

13. Pino, S., Pollock, L., Chandrasekaran, S.: Exploring translation of OpenMP to OpenACC 2.5: lessons learned. In: IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 673–682. IEEE (2017)
14. Wolfe, M.: Compilers and more: OpenACC to OpenMP (and back again), June 2016. https://www.hpcwire.com/. Accessed 29 June 2016