Investigating the HIP programming model with regards to portability and performance portability

Niklas Kerscher
Department of Computer Science
Technical University Munich
Munich, Germany
niklas.kerscher@tum.de

July 4, 2022

***Abstract*—While modern HPC systems are being dominated by NVIDIA GPUs, new vendors such as AMD and Intel are entering the field, which creates the problem of software portability across different GPU architectures. Therefore portable programming frameworks were introduced, to allow code to be portable across different systems. Ideally, the performance level of the same application is portable across the different architectures and the porting process is fast and work efficient. Tackling these challenges, AMD introduced the Heterogeneous-compute Interface for Portability (HIP) [1], which is an open-source C++ runtime API and kernel language. It provides compatibility between CUDA and ROC and is designed to deliver close to native performance on CUDA machines while exposing additional low-level hardware features [2]. In this work, the performance portability of HIP and the possibilities and effort of its portable features are described and evaluated. In addition, the challenges regarding this research are presented and the future role of HIP and other parallel programming models like Kokkos, SYCL, etc. is discussed. These models are mostly based on the underlying concepts of the low-level OpenCL programming model and provide further code portability for all supported architectures, while also incorporating all features of high-level open-standard parallel programming frameworks. HIP performed equally well, compared to other competing frameworks and porting code using hipify is easy, although minor code modifications might be necessary.**

***Index Terms*—HPC, HIP, Performance Portability, CUDA**

I. Introduction

Taking a look at the current top ten spots on the TOP500 list of supercomputers [3], as of November 2021, eight of them are built on heterogeneous systems. Also, more than 20% of the machines listed in Top500 rely on GPUs for their computing power and indicators point to an increase in this number as upcoming supercomputer projects, will also feature GPU-based models [4]. As stated in Cade Brown et.al.'s paper [4] another factor for GPU designs in HPC clusters, is the use of Artificial Intelligence and Machine Learning, both of which require a boost in low-precision calculations [4]. NVIDIA's latest Tensor Cores, able to reach 624 TFlop/s, have been developed specially for these use cases. For a couple of years, GPUs have been at the forefront, when it comes to choosing accelerators for supercomputers. They meet performance requirements while keeping an acceptable power consumption profile. Currently still in development, Frontier[1] and El Capitan will be using AMD's EPYC CPU and Radeon Instinct GPU architecture [5,6] to achieve performance levels, exceeding the exascale limitation. This also means that AMD's ROCm open compute platform will be integrated into both machines. Other machines in the supercomputer race include the Intel-based Aurora system and Japan's Fugako computer, which uses chips based on the ARM-64 architecture. Furthermore many current HPC clusters use a mix of Intel, AMD and NVIDIA processors. This vendor and architecture heterogeneity highlights the need for code and performance portability. In the past years the HPC community developed and adopted many of these so-called parallel programming frameworks, to tackle the posed challenges. Multiple frameworks, which work independently of the underlying architecture, while maintaining performance and portability, include Kokkos, SYCL, OpenCL, HIPCL, HIP and others [7]. However these approaches only present wrappers for the underlying vendor-specific solutions like CUDA and HIP, for NVIDIA and AMD systems. This plethora of standards has led to a lack of well-adopted standards in the HPC community for coding functionality and performance portability on these different systems [4].

II. Background

In this paper, we focus on HIP, which is a C++ based programming model. As of June 2022 not much literature has been published, that benchmarks the performance and performance portability of just HIP, as it is mostly used to draw a bridge between NVIDIAs CUDA and AMDs ROCm platform. As pointed out by Amanda S. Dufek et.al.'s paper [7], the performance of different models like SYCL, OpenCL and others has already been examined by different authors and they concluded that all the described models delivered equivalent performance in relation to their theoretical peak memory bandwidth. Also, different SYCL compilers have been developed and tested on their corresponding backends (which includes HIP), however, these did not test HIP performance when porting code from CUDA to the ROCm platform or vice versa. Still, there has been promising research as mentioned in [7], done by Homerding and Tramm, that showed competitive

[1]During the development of this paper the Frontier system at ORNL went online, taking the number one spot in the latest June 2022 edition of Top500.

performance levels when comparing the implementation of two HPC mini-apps on hipSYCL[8], an implementation of SYCL built on top of NVIDIA CUDA/AMD HIP [9], and a native CUDA implementation.

A. Performance portability in HPC-systems

There has been a trend from CPU-heavy supercomputers, to those that are mainly driven by GPU computing power. GPUs in symbiosis with CPUs have shown, that their provided performance applies to many different applications like industry, cloud and leadership computing facilities that use these machines for nuclear weapon modelling and other research areas [5]. The GPUs advantageous parallelism delivers much higher throughput and better energy efficiency than current CPUs [10]. Underlining this movement, the Summit supercomputer, currently America's strongest computer and only second to the Fugaku in the world ranking [3] as of November 2021, gains 97% of its 200 petaFLOP output (theoretical peak performance) from the roughly 27000 GPUs [11], which shows the reliance on these processing units nowadays.
The primary vendor for GPUs has been NVIDIA for the most part [11], and their CUDA programming model has been widely used in all highly optimized HPC GPU kernels. As of November 2021 over 98% of all accelerators in HPC systems come from NVIDIA [3]. Nonetheless other vendors such as AMD and Intel have struck deals to provide hardware and software for even stronger supercomputers. Adding to this, the introduction of ARM on certain HPC systems, like the Fugaku, poses even more challenges to developers. This creates the problem of code and performance portability as described in section I.

B. The Heterogeneous-compute Interface for Portability

Developed especially for use with C++, the HIP programming framework comes with its own unique toolbox and features, to ease development on HPC systems. The open-source framework offers coding in a single-source C++ programming language and includes features like templates, C++11 lambdas, classes and namespaces, while allowing developers to use the development environment on each target platform [12]. Not supported by the HIP API are textures, dynamic parallelism (CUDA 5.0), managed memory (CUDA 6.5) and graphics interoperability with OpenGL or Direct3D and other features [13]. For more information the AMD HIP Programming Guide [12] and the HIP-FAQ [13] can be consulted. Although HIP is mostly viewed as a one-time approach for porting CUDA software to run on AMD GPUs, it can also be used extensively as a portability layer for simultaneously targeting both AMD and CUDA systems [14].
Programmers working on AMD and NVIDIA GPUs also use the OpenCL programming framework, which is similar to the HIP API. However HIP offers various advantages, like its C++ framework language, which is the same as CUDAs, allowing for the mixing of host and device C++ code in source files and providing additional functionality, as stated before. Also, HIP is less verbose than OpenCL and has a similar user-experience for CUDA-used developers.
Additionally, hipSYCL has been developed, to connect the SYCL implementation ecosystem to existing toolchains like HIP and CUDA. It targets any CPUs via OpenMP, NVIDIA GPUs via CUDA, AMD GPUs via HIP/ROCm and Intel GPUs via oneAPI Level Zero and SPIR-V. The hipSYCL compiler can compile CUDA and HIP code in the same source file and after compilation, creates a single binary, that can run on all backends with the appropriate clang distributions [15].

C. Porting CUDA to HIP

HIP also includes the hipify tool [16], which allows users to convert existing CUDA code to HIP. This can be achieved by either using hipify-clang or hipify-perl.

1) hipify-clang: Hipify-clang offers a source-to-source translator, that uses the clang/LLVM compiler front-end. It creates an abstract syntax tree, that is traversed by transformation matchers and after applying all of them, the HIP source code is produced [16]. The clang version can traverse very complicated constructs and parse them successfully, or report potential errors, while providing safe support for future CUDA versions, as it has a strong subset of the functionality provided by CUDA. However, the fact that any error will be reported, means that for a successful translation, the CUDA code has to be correct. Also, hipify-clang needs all the necessary include and define headers to work successfully [16] and, as reported in [14], hipify-clang did not work with CUDA version 10.1 and it did not consider, at least partially, preprocessor macros.

2) hipify-perl: Hipify-perl is an auto-generated perl-based script, that uses a text-based search and replace algorithm and therefore relies heavily on regular expressions [16]. Its advantages include ease of use, not having dependencies on third party tools and not checking the input CUDA source code for correctness and therefore always producing a successful parse. However, hipify-perl still has difficulties in supporting and is not able to transform various code constructs, like macros expansion, namespaces, templates etc. Therefore it may be advisable to use hipify-clang for more complicated code-bases, but for CUDA code with basic functionality, [14] commented, that hipify-perl seemed to work even better than its counterpart.

III. Portability and Performance Portability Evaluation

Modern parallel programming frameworks have to provide the programmer with the necessary portability possibilities, for running code on different architectures, while also introducing little overhead among the different vendor-specific programming models, like CUDA, ROCm or SYCL/DPC++.

A. Approach

Most papers regarding HIP only focus on its performance as a backend solution for running frameworks like Kokkos and

SYCL on AMD machines. Still, these results are sufficient in painting a good enough picture of HIP's performance and portability. The chosen approach consists of taking different papers benchmarking results and the corresponding conclusions and projecting them to this paper. The variety of papers using HIP in backend solutions, or those evaluating its actual performance on CUDA-based systems, offer enough diversity to draw valid conclusions. In addition, the provided sources already nearly cover the available research regarding this topic. For future research, own benchmarks on CUDA-based HPC clusters, with the HIP framework, would be applicable.

B. Performance Portability

In this paragraph, we try to evaluate the performance of the HIP programming framework. This means we have to look at different papers that have studied its core features. As stated in section II, these include the porting of CUDA code to HIP via hipify and running HIP code on AMD and NVIDIA systems. Consequently, we need to compare the performance of applications written natively on CUDA, which are then converted using hipify, on systems with NVIDIA and AMD GPUs. These benchmarks then draw a picture of the eventual overhead produced by hipify to run native CUDA code on AMD GPUs and the other way round. Following up, in section III-C different aspects of the porting process will be evaluated, like ease of use and error affinity.

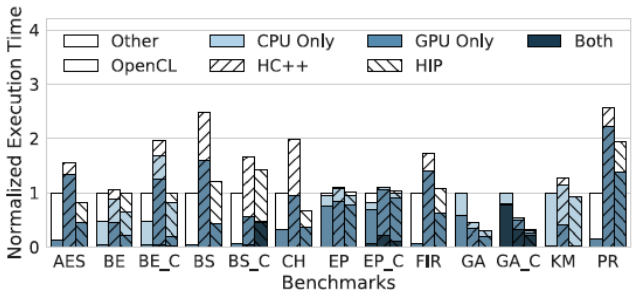

Fig. 1. Performance comparison of different frameworks running on the ROC platform. The bars in each group, from left to right, represent OpenCL, HC++, and HIP, respectively. C denotes collaborative execution. Figure from [10, Adapted under "free use"]

1) HIP, HC++ and OpenCL on the ROCm platform: Starting with [10], which examined HIP in different benchmark settings to evaluate its performance, we get a good idea of HIPs performance. First collaborative performance on multiple platforms and programming models was tested. This was achieved using the Hetero-Mark and DNNMark benchmark suites. Hetero-Mark provides a rich set of CPU-GPU communication and collaborative patterns, while DNNMark is used for Deep Neural Networks targeting GPUs. More information, on the exact benchmarks used, can be found in [10, Tab. 1]. Figure 1 shows the performance of the same workloads, implemented using different frameworks all running on the AMD ROCm platform. As stated in [10] the used AES, CH and FIR benchmarks are very memory heavy, which is mirrored by the results, showing little actual GPU time and more "other" time, due to the bigger memory transfers [10]. For these benchmarks, the HC++ implementations take the longest, while HIP manages to perform the best. This underlines the performance advantage HIP has on its proprietary platform ROCm. The GA and PR benchmarks show a slightly different picture, although the performance difference for PR is mainly attributed to the different kernel implementations, which are necessary for each framework. Therefore the results are not further comparable [10]. The GA benchmark on the other hand has the same kernel implementation in both OpenCL and HIP, however, the number of instructions executed in both kernels differs greatly. While HIP is only using around 160K ALU instructions, the OpenCL implementation needs around 270K ALU instructions. This indicates that the HIP kernel compiler is more efficient than the OpenCL one [10]. Concluded from the bench-marking results the HIP kernel compiler offers the best proprietary solution on the ROCm platform, performance-wise, while also handling an efficient ALU instruction footprint.

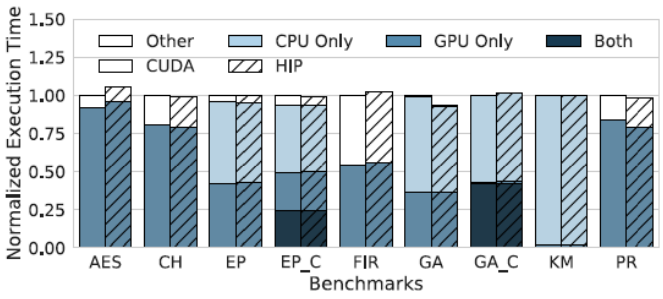

Fig. 2. Performance of HIP and CUDA running on an NVIDIA GTX 1080. The bars in each group, from left to right, represent CUDA and HIP, respectively. Figure from [10, Adapted under "free use"]

2) Performance of HIP and CUDA on an NVIDIA system: Seen in figure 2 is a benchmark of HIP's main feature, porting CUDA code to HIP so that a unified code basis can be run on CUDA and ROCm platforms. Here code was ported to HIP, run on an NVIDIA system using a GTX 1080, and run natively with CUDA on the same system [10]. The different benchmark results show that HIP adds no noticeable overhead to the workloads and the execution time varies from $0.93\times$ to $1.05\times$ of the corresponding CUDA implementation [10]. The paper concludes, that independent programming frameworks improve the portability, while not leaving any performance on the table, with HIP having the highest average performance among OpenCL and HC++ [10]. Unfortunately, this and [2] are, more or less, the only available papers, researching HIPs performance on NVIDIA systems in comparison to native code. Although the results look promising, future research has to be done, to define how useful HIP is on CUDA systems. Should the results be replicable, code bases written in HIP could mostly replace CUDA, as HIP is portable across architectures.

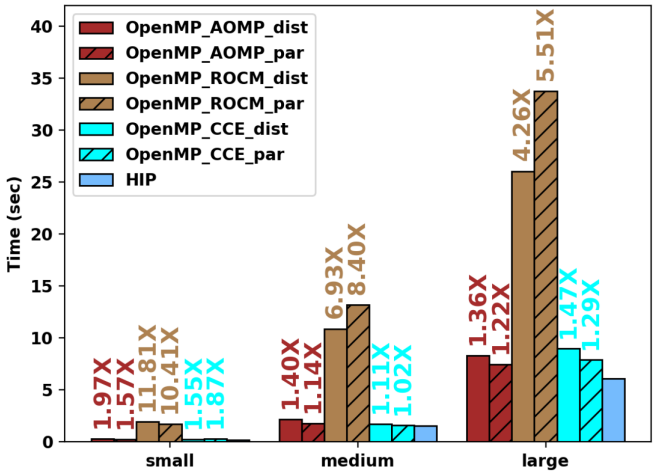

Fig. 3. Performance of OpenMP target offload on AMD MI100 GPUs for nruns set to 10. Indicated are the exact times in seconds, and the slowdown (×) compared to the HIP version for each input (small, medium, and large ligand). Figure from [17, Adapted under "free use"]

3) Slowdown of OpenMP target offload compared to HIP: Another approach from [17] evaluated the performance of AMD MI100 GPUs using HIP and OpenMP offload versions of a molecular docking application using three different compilers: HPE-Cray's Cray Compiling Environment (CCE), AMD's ROCm compiler and the OpenMP focused LLVM-Clang based AOMP compiler. Although for medium inputs a near-identical performance could be achieved between HIP and CCE_dist, especially for small and large inputs, the slowdown amounted to $1.55\times$ and $1.47\times$ for CCE_dist and $1.87\times$ and $1.29\times$ for CCE_par, respectively, as seen in figure 3 [17]. The author states that this is within an acceptable range for portable framework solutions, although the performance could be improved upon. A similar picture presents itself for the OpenMP AOMP compiler, which has a minimal slowdown of $1.22\times$ [17]. Therefore the performance of the native HIP version of the application, could not be matched or exceeded by the OpenMP programming model. As shown in [17, Fig. 10] the overall performance of HIP on the AMD system was on-par with the respective CUDA version, although for large inputs the application performs better on NVIDIA's native system. In a rapidly changing landscape for HPC clusters with different vendors entering the market, the use of a performance portable version of an application may become applicable, as the performance directives for these applications are being met on the different GPU architectures [17]. Therefore the author concludes, that maintaining only one version of a program may be a valid solution in the future. This also underlines the good performance of HIP and the use cases it manages to satisfy.

4) Simple kernels performance using HIP, OpenCL, Kokkos and Julia: For another performance benchmark of HIP on AMD GPUs, Wei-Chen Lin et.al.'s paper [18] compared it to Kokkos, OpenCL and two different Julia implementations of simple kernels. These benchmarks compare the performance of five different memory-bandwidth bound kernels, using various systems. This benchmark specifically, employs three different systems with AMD GPUs, the Instinct MI100, the Instinct MI50 and the Radeon VII. For the research of this paper, the results achieved by the Radeon GPU are less relevant, as it is a consumer GPU. Although the paper interprets its results with regards to the performance of Julia in HPC systems, the recorded outcomes provide a sufficient overview of HIPs performance on AMD GPUs, since we get a comparison of different parallel programming frameworks. The results in figure 4 show that HIP performance in Copy and Mul kernels is very similar and for Add and Triad kernels, HIP and OpenCL managed to achieve nearly even results with $76.0, 74.3, 78.5$ and $76.9, 74.6, 77.5$ peak memory bandwidth percentage (for the Add kernel) respectively, while Kokkos stayed only a little behind achieving $75.8, 71.6, 75.2$ peak memory bandwidth percentage. Only in the Dot kernel implementation OpenCL outperformed both, with HIP taking the middle spot, between the two others, performance-wise.

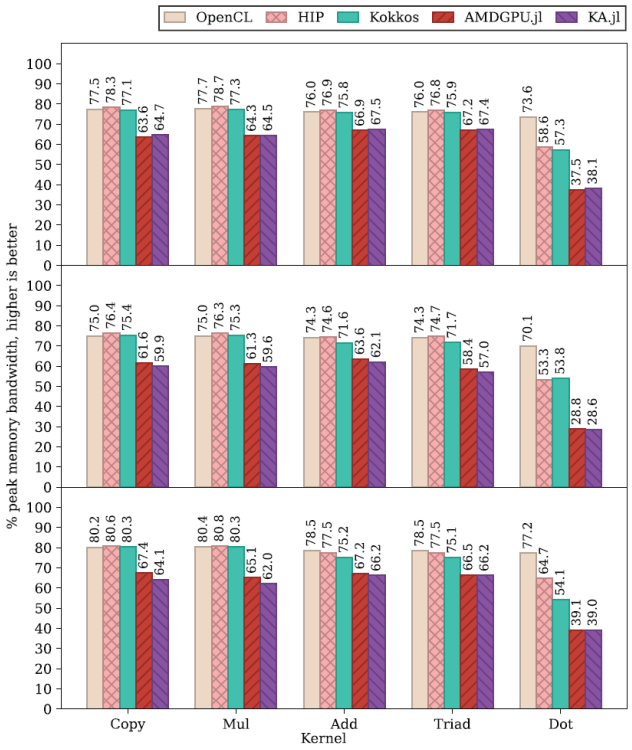

Fig. 4. BabelStream AMD GPU results. Figure from [18, Adapted under "free use"]

Summing up the performance evaluation of HIP, it managed to show results that matched or even exceeded other parallel programming frameworks, when being run on AMD systems [11,18]. Although the performance of HIP on systems by NVIDIA was not tested thoroughly, [10] managed to show on-par performance with native CUDA code. Shedding more light

on HIPs performance on CUDA-based machines, [2] presents more in-depth graphs using GINKGO SpMV kernels and the Conjugate Gradient Solver employing the Sellp SpMV kernels for the Kylov subspace generation. The left and right sides of [2, Fig. 4] only differ in the use of atomic operations. The benchmark showed better performance for the native CUDA implementations, although HIP provided little overhead in most problems [2]. Interestingly for some problems, CUDA was a lot faster than HIP, and for non-atomic operations, HIP also managed to outperform CUDA significantly on its native platform in some cases. However, the mean and variance of those performance outliers [2, Fig. 6] emphasize, that they are inconsequential, as 90% of the test cases showed less than 10% performance difference. Therefore the performance overhead of HIP is minor, compared to CUDA implementations on NVIDIA systems. Looking at [10, Fig. 6], the memory management in HIP also delivers greater performance than its corresponding counterparts in this benchmark. Even though it is not intended to provide a universal solution for porting code bases to any GPU platform, like Kokkos, SYCL and OpenMP/CL, there have been developments in integrating these frameworks with HIP. Highlighted in [19], HIPCL enables the possibility of porting HIP code to OpenCL platforms, which increases the portability as OpenCL platforms are independent of the underlying vendor-specific architecture. From [19, Fig. 1, Fig. 2] the performance of HIPCL and OpenCL is shown to be identical and the introduction of HIPCL seems to add negligible overhead. This concludes that HIP offers a great choice performance-wise, for a parallel programming framework for HPC systems. Although its use is limited to running on CUDA and ROCm systems, there are possibilities of increasing this portability, while maintaining good performance levels.

C. Portability

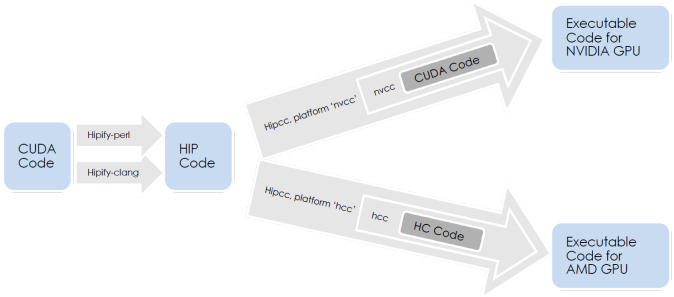

Fig. 5. Converting CUDA Code to Executable Via HIP. Figure from [14, Adapted under "free use"]

The portability of HIP is limited to CUDA (Fig. 5) and ROCm systems, although there are possibilities of extending this with kernel libraries like HIPCL and hipSYCL [8,20]. While other approaches, also include portability to vendors like Intel, or ARM systems, nearly all GPUs in HPC systems worldwide are provided by NVIDIA and AMD, with AMD only recently setting foot in this market. Therefore the portability offered by HIP is satisfying for most use cases, even though many papers use frameworks on top of HIP to benchmark the performance of applications [7,9,19,21]. Code from CUDA can be ported to HIP in a relatively automated fashion using hipify, as illustrated in figure 5. The HIP code can then be executed on either AMD or NVIDIA GPUs.

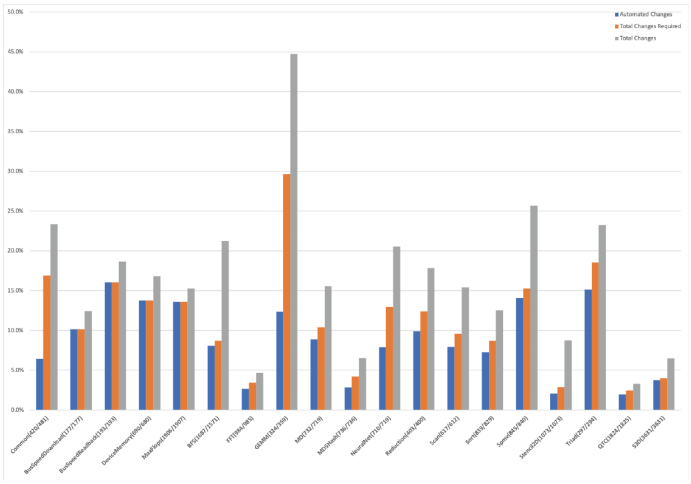

Fig. 6. Percentage of lines of CUDA version of SHOC benchmark programs, that were automatically changed, required manual changes and those that were desirably changed. The labels are the component names (CUDA LOC/HIP LOC). Figure from [14, Adapted under "free use"]

1) Porting errors using hipify: The porting effort using hipify is relatively manageable, as the philosophy of HIP was to keep the language close enough, regarding function names, to CUDA, so that conversion is easier [21]. This reduces porting errors, as described in [11]. Although the conversion process is easy, code modifications may still be needed for the code to work properly. As can be seen in figure 6, the manual alterations required for a working code-base vary from nearly 2.5% to as much as 29%, which means the developer may need to spend extra time polishing the parsed code, for it to function unhindered with the HIP framework. In this example, these modifications can be attributed to header alterations and code-changes from cuBLAS to hipBLAS, which is a basic linear algebra subprograms marshalling library, with support for cuBLAS (NVIDIA) and rocBLAS (AMD) as backend [22]. Also, most of the desired manual changes include name changes from CUDA to HIP in variable names, class names, macros and filenames, which are only important for the naming consistency of the project, but not vital to a working application [14]. The fact that most of these modifications are minor, make hipify a great tool for code conversion and it shows that HIP's portability, although limited to NVIDIA and AMD, works great for both architectures.

2) Portability using hipSYCL and HIPCL: There has also been a development, trying to integrate the SYCL and OpenCL toolchain into the HIP API. As explained in [8], hipSYCL is a modern SYCL implementation that targets CPUs and

GPUs. This brings maintenance and stability advantages and allows for maximum interoperability with existing compute platforms. HIPCL is also using the HIP API, but extends portability to devices supporting OpenCL and SPIR-V, and thus provides a portability path from CUDA to OpenCL [20]. So far, there has been little research exploring the porting process and potential errors for both toolchains.

IV. Discussion

HIP provides a good solution for running code on the CUDA or ROCm platform, while also seemingly not being limited by performance hurdles. Still, there is currently little research focusing solely on HIP's performance. It would be advisable to benchmark HIP, by porting native CUDA code via hipify to HIP and then running both on the same NVIDIA machine. Also, the accuracy and performance of converting code from CUDA to HIP is an interesting topic, which needs further research, as [11] found that CUDA codes that use numerous hardware-level optimizations may be unusable in HIP, because direct translations may not exist. The authors stated that manual code discovery of different syntax, usage rules and differences in the default behaviour of functions was required when porting code via hipify [11]. However, this still posed to be less time-consuming than rewriting code, when going from OpenCL to CUDA or Kokkos [11]. With multiple parallel programming frameworks competing for general recognition and adoption in the HPC community, this opens up a new challenge of porting code between the different frameworks. Modern applications written for specific HPC clusters may be able to run on various architectures, however, the adoption of new architectures from new companies like Intel will take time. In this transition phase, translating code from different frameworks could be necessary. In [11] it is estimated that these translations, in this case porting OpenCL to CUDA or Kokkos, take $2-3$ times more worker hours, than the conversion to HIP. This can be explained by HIP, allowing other GPU vendors, to use a similar API to CUDA, which makes conversion faster, according to [11]. Therefore the development of universal translation frameworks for parallel programming models would be an interesting topic in future research if there is no agreement on a universal standard for programming models. The lack of benchmarks for HIP's performance can also be attributed to the rather recent entry of AMD into the GPU market for HPC systems. Once Frontier [6] will go online, this might change, as the exascale supercomputer is set to become the most powerful one yet while relying on AMD CPU and GPUs. This means that the ROCm platform will become more significant, which should translate to more research interest in HIP and hipify.

V. Summary and Outlook

In this paper the portability and performance portability of HIP was presented and evaluated. Although the used sources mostly offered an insufficient way of benchmarking HIP, they portrayed nicely its different facets and use cases. All together the different benchmarks gave a good outlook on how HIP performs in real-world applications. The results indicated that HIP offers similar performance to native CUDA implementations and as a backend solution, with frameworks like Kokkos and SYCL, it provided great performance portability across different platforms. Papers like [17] highlighted that it can make a significant difference which backend solution is chosen to enable truly performance portable programs. Overall HIP showed the most consistent performance and also managed to outperform native CUDA code on NVIDIA machines in certain cases.

With the launch of supercomputers like El Capitan and Frontier on the horizon, that are based on Intel and AMD architectures, it is to be expected that the research in portable programming frameworks increases. In addition papers like [2,11] promised future research regarding HIP and its use as a backend solution. The outlook for the future of performance-portable solutions is looking promising as new GPU architectures and programming frameworks enter the HPC ecosystem and disrupt NVIDIA's near-monopoly on GPUs for HPC systems.

References

[1] AMD, "HIP," [Online; accessed 31-May-2022], 2022. [Online]. Available: https://github.com/ROCm-Developer-Tools/HIP

[2] Y. M. Tsai, T. Cojean, T. Ribizel, and H. Anzt, "Preparing Ginkgo for AMD GPUs – A Testimonial on Porting CUDA Code to HIP," in *Euro-Par 2020: Parallel Processing Workshops*, B. Balis, D. B. Heras, L. Antonelli, A. Bracciali, T. Gruber, J. Hyun-Wook, M. Kuhn, S. L. Scott, D. Unat, and R. Wyrzykowski, Eds. Cham: Springer International Publishing, 2021, pp. 109–121.

[3] Top500, [Online; accessed 29-May-2022], 2021. [Online]. Available: https://www.top500.org/lists/top500/2021/11/

[4] C. Brown, A. Abdelfattah, S. Tomov, and J. Dongarra, "Design, Optimization, and Benchmarking of Dense Linear Algebra Algorithms on AMD GPUs," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–7.

[5] R. Smith, "El Capitan Supercomputer Detailed: AMD CPUs & GPUs To Drive 2 Exaflops of Compute," [Online; accessed 29-May-2022], 2020. [Online]. Available: https://www.anandtech.com/show/15581/el-capitan-supercomputer-detailed-amd-cpus-gpus-2-exaflops

[6] M. L. McCorkle, "U.S. Department of Energy and Cray to Deliver Record-Setting Frontier Supercomputer at ORNL," [Online; accessed 29-May-2022], 2019. [Online]. Available: https://www.ornl.gov/news/us-department-energy-and-cray-deliver-record-setting-frontier-supercomputer-ornl

[7] A. S. Dufek, R. Gayatri, N. Mehta, D. Doerfler, B. Cook, Y. Ghadar, and C. DeTar, "Case Study of Using Kokkos and SYCL as Performance-Portable Frameworks for Milc-Dslash Benchmark on NVIDIA, AMD and Intel GPUs," in *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2021, pp. 57–67.

[8] Heidelberg University, "HIPSYCL," [Online; accessed 01-July-2022], 2022. [Online]. Available: https://github.com/illuhad/hipSYCL

[9] B. Homerding and J. Tramm, "Evaluating the Performance of the HipSYCL Toolchain for HPC Kernels on NVIDIA V100 GPUs," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3388333.3388660

[10] Y. Sun, S. Mukherjee, T. Baruah, S. Dong, J. Gutierrez, P. Mohan, and D. Kaeli, "Evaluating Performance Tradeoffs on the Radeon Open Compute Platform," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2018, pp. 209–218.

[11] M. Thavappiragasam, A. Scheinberg, W. Elwasif, O. Hernandez, and A. Sedova, "Performance Portability of Molecular Docking Miniapp On Leadership Computing Platforms," in *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2020, pp. 36–44.

[12] AMD, "AMD HIP Programming Guide," [Online; accessed 30-June-2022], 2021. [Online]. Available: https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html
[13] AMD, "HIP-FAQ," [Online; accessed 30-June-2022], 2021. [Online]. Available: https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-FAQ.html#hip-faq
[14] P. C. Roth, "Experiences with the Heterogeneous-compute Interface for Portability (HIP) on OLCF Summit," [Online; accessed 30-June-2022], Oak Ridge National Laboratory, 2019. [Online]. Available: https://www.olcf.ornl.gov/wp-content/uploads/2019/10/Roth-HIP-on-Summit-20191009.pdf
[15] A. Alpay and V. Heuveline, "SYCL beyond OpenCL: The Architecture, Current State and Future Direction of HipSYCL," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3388333.3388658
[16] AMD, "hipfy," [Online; accessed 31-May-2022], 2022. [Online]. Available: https://github.com/ROCm-Developer-Tools/HIPIFY
[17] M. Thavappiragasam, W. Elwasif, and A. Sedova, "Portability for GPU-accelerated molecular docking applications for cloud and HPC: can portable compiler directives provide performance across all platforms?" *arXiv e-prints*, p. arXiv:2203.02096, Mar. 2022.
[18] W.-C. Lin and S. McIntosh-Smith, "Comparing Julia to Performance Portable Parallel Programming Models for HPC," in *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2021, pp. 94–105.
[19] M. Babej and P. Jääskeläinen, "HIPCL: Tool for porting CUDA applications to advanced OpenCL platforms through HIP," *Proceedings of the International Workshop on OpenCL*, Apr. 2020, IWOCL / SYCLcon ; Conference date: 27-04-2020 Through 29-08-2020.
[20] Tampere University, "HIPCL," [Online; accessed 01-July-2022], 2021. [Online]. Available: https://github.com/cpc/hipcl
[21] Z. Jin and J. Vetter, "Evaluating CUDA Portability with HIPCL and DPCT," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2021, pp. 371–376.
[22] AMD, "hipBLAS," [Online; accessed 30-June-2022], 2022. [Online]. Available: https://github.com/ROCmSoftwarePlatform/hipBLAS