Evaluating CUDA Portability with HIPCL and DPCT

Zheming Jin and Jeffrey Vetter
Oak Ridge National Laboratory
jinz@ornl.gov

***Abstract*— HIPCL is expanding the scope of the CUDA portability route from an AMD platform to an OpenCL platform. In the meantime, the Intel DPC++ Compatibility Tool (DPCT) is migrating a CUDA program to a data parallel C++ (DPC++) program. Towards the goal of portability enhancement, we evaluate the performance of the CUDA applications from Rodinia, SHOC, and proxy applications ported using HIPCL and DPCT on Intel GPUs. After profiling the ported programs, we aim to understand their performance gaps, and optimize codes converted by DPCT to improve their performance. The open-source repository for the CUDA, HIP, and DPCT programs will be useful for the development of a translator.**

Keywords—CUDA, HIP, OpenCL, DPC++, CUDA Portability

I. Introduction

NVIDIA CUDA [1], which was introduced in 2007, has successfully enabled the use of a graphics processing unit (GPU) as a programmable general-purpose computing device. However, CUDA is a proprietary programming model for NVIDIA GPUs. OpenCL, on the other hand, is an open standard maintained by the Khronos group with the support of major graphics hardware vendors as well as personal computer vendors interested in offloading computations [2, 3]. Hence, OpenCL offers programming portability across a wide range of software and hardware for GPUs, multi-core processors (CPUs), and other accelerators.

In contrast to OpenCL which is based on the C programming language, SYCL is a specification which defines a single-source C++ programming layer on top of OpenCL [4]. Hence, SYCL allows a developer to create applications and libraries with C++ without using OpenCL host and kernel languages. The goals of the single-source programming model and the support for C++ features are to improve programming productivity and performance portability [5, 6, 7, 8, 9, 10, 11].

NVIDIA and AMD have been driving most of the discrete GPU market. Hence, there are significantly more CUDA or HIP applications available than those implemented in OpenCL and SYCL. On the other hand, the OpenCL application-programming interface (API) is a lower-level architecture compared to the commonly used CUDA API, thus requiring more time and effort to develop an OpenCL host program for the management of device, memory, and kernel execution. Such process is often tedious and error prone.

Acknowledging CUDA's established presence in high-performance computing and alleviating the pain of manual development of OpenCL programs, researchers have been striving for a portability-enhancing path for a wider set of platforms [12, 13, 14, 15]. Towards the goal of portability enhancement, in this paper we evaluate the performance of CUDA applications ported using HIPCL [16] and the Intel DPC++ Compatibility Tool (DPCT) [17] on Intel computing platforms. While there are other translators for porting CUDA codes [18, 19, 20, 21], they have not been under active development for a while. HIPCL and DPCT are ready for porting the applications in our study. After profiling the codes ported using HIPCL and DPCT, we aim to understand the cause of their performance gaps. Furthermore, we optimize the codes converted by DPCT to improve their performance on the GPUs. We summarize the contribution of the paper as follows.

- Open access to the applications for evaluating CUDA portability on GPUs
- Performance evaluation of the applications ported using HIPCL and DPCT on Intel GPUs
- Analysis and optimization of the applications ported with DPCT for performance improvement

The rest of the paper is organized as follows. Section II introduces different programming models (HIP, HIPCL, SYCL, DPCT) and the architecture of an integrated GPU. Section III describes the evaluation and optimization of the applications on the GPUs. Section IV summarizes related work, and Section V concludes the paper.

II. Background

Figure 1 shows the method of evaluating CUDA portability with DPCT and HIPCL. We will explain the flow in the following sections.

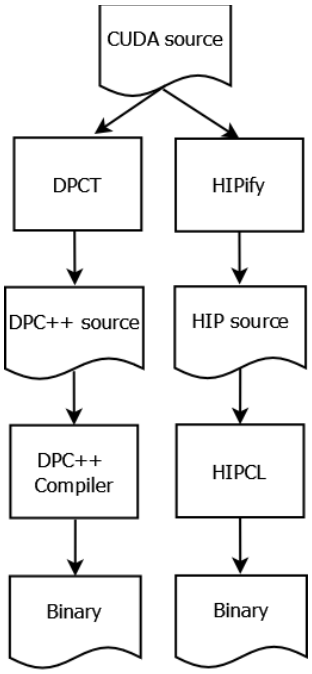

Fig 1. The flow of evaluating CUDA portability

A. HIP

Heterogeneous-compute Interface for Portability (HIP) is an API written in the C++ programming language for developers to run applications on AMD and NVIDIA GPUs [22]. The HIP philosophy was to make the HIP language close enough, in terms of the function names, to CUDA that the porting effort is generally straightforward and simple. This reduces the potential for porting errors, making it easy to automate the translation. The goal of HIP is to run a ported program on both platforms with little manual intervention [23]. A CUDA source can be converted to a HIP source in a largely automated fashion using the HIPify-Clang utility in AMD's developer tools [24].

B. HIPCL

HIPCL is a new library for running HIP programs on devices supporting OpenCL and SPIR-V [25]. Since the APIs of CUDA and HIP are close enough and the porting effort is typically straightforward, HIPCL provides a portability path from CUDA to OpenCL. HIPCL consists of three components: Clang, the runtime library, and the kernel library [16]. A patched Clang is required to compile a single-source C++ program to a Linux ELF binary which bundles device code in SPIR-V and host code. The runtime library implements HIP API functions, which are called in a host program, by mapping them to OpenCL API equivalents. The kernel library implements the HIP math API by using the OpenCL C math built-ins and Intel-specific OpenCL extensions. The library requires OpenCL version 2.0 for features such as virtual address space and subgroup functions. Since AMD and NVIDIA do not support SPIR-V currently, HIPCL has been mainly tested on Intel GPU devices.

C. SYCL

The design of SYCL allows for the combination of the performance and portability features of OpenCL and the flexibility of using high-level C++ abstractions [4]. Most of the abstraction features of C++, such as templates, classes, and operator overloading, are available for data-parallel functions (i.e., kernels) executed on a device such as a GPU. Some C++ language features, such as virtual functions, virtual inheritance, throwing/catching exceptions, and runtime type-information, may be disallowed inside kernels due to the capabilities of the underlying standard. These features are available outside the kernel scope.

D. DPCT

The tool can migrate a CUDA program to a data parallel C++ (DPC++) program. DPC++ extends SYCL with additional extensions and provides support for a variety of OpenCL devices [26]. The tool claims that it can port both CUDA kernels and library API with 80% – 90% of CUDA codes automatically migrated to DPC++ codes [17]. When migrating CUDA codes, comments are added in DPC++ codes to suggest that users modify migrated codes when they can be optimized away or are not transformed automatically due to the constraints of the programming model, translator, or computing capabilities. An example of migrating a CUDA program is shown in [17]. While the tool is a component of the Intel oneAPI toolkit, a standalone version is also available.

DPCT can convert a CUDA program to a DPC++ program in which memory management migration is implemented using the explicit and restricted unified shared memory (USM) extension or the DPCT header files. By default, the generated codes rely on explicit and restricted USM extension for memory management migration. The empirical results of migrating over many CUDA applications show that the default implementation can generally achieve higher performance and portability coverage [27]. Hence, we will choose the USM-based codes for evaluating the performance of the ported applications. For ease of description, we will refer to DPC++ programs, which are generated from CUDA programs using DPCT, as DPCT applications, codes, or programs.

E. Intel integrated GPUs

In the architecture of an Intel integrated GPU [28], a GPU connects to CPU cores via a ring interconnect, and they share a main memory with CPU cores. To reduce data access latency from a main memory, a GPU maintains a memory hierarchy comprised of register files, instruction caches, and data caches. Some products include an embedded dynamic random-access memory behind a last-level cache to further reduce latency to system memory for higher effective bandwidth. The building block of the graphics compute architecture is an execution unit

Table 1. Characteristics of the 18 Applications ($M = 2^{20}$)

Name	Domain	#Kernels	Problem size
b+tree [29]	Database search	2	1 million keys
backprop [29]	Pattern recognition	2	65536 keys
bfs [29]	Graph traversal	1	1 million vertices
cfd [29]	Fluid dynamics	5	97047 elements
gaussian [29]	Linear algebra	2	4096×4096 matrix
heartwall [29]	Medical imaging	1	104 frames
hotspot3D [29]	Physics simulation	1	512×512 points
hybridsort [29]	Sorting	7	50 million numbers
kmeans [29]	Data mining	2	494020 points and 34 features per point
particlefilter [29]	Medical imaging	4	400000 points
nw [29]	Bioinformatics	2	2048×2048 data points
srad [29]	Image processing	6	512×512 data points
lud [29]	Linear algebra	3	8192×8192 points
sort [30]	Sorting	3	16M numbers
md5hash [30]	Cryptography	1	10M keyspace
fft [30]	Linear algebra	2	16M complex numbers
s3d [30]	Combustion simulation	27	16×16×16 grid
miniFE [31]	Unstructured grids	9	128×128×128 grid

(EU). It is a combination of simultaneous multi-threading and fine-grained interleaved multi-threading. In general, each EU can run seven threads concurrently to hide memory access latency. Arrays of EUs are organized as a subslice. The number of EUs per subslice depend on the generation of compute architecture. Each subslice contains a thread dispatcher unit and supporting instruction caches. Subslices are grouped into slices. A slice integrates additional logics for thread dispatch routing, banked L3 data cache, banked shared memory, and fixed function logic for atomics and barriers.

III. Experiment

A. Applications

Table 1 lists the applications from Rodinia [29], SHOC [30], and proxy-apps [31] used in our work. These applications, which cover a variety of scientific domains, have been widely used for performance evaluation [32, 33, 34, 35, 36]. The number of kernels in the table indicate the number of distinct kernels which are executed at least once on a device. The problem sizes of the selected Rodinia applications are larger than or equal to the original sizes. For the SHOC programs, we choose one of the four problem sizes specified in the original programs. The CUDA, HIP, and DPCT implementations are available in the public repository [27].

B. Setup

We evaluate the applications on two computing systems. The first one (System 1) has an Intel Xeon E3-1284L v4 CPU running at 2.9 GHz. The CPU has four cores and each core supports two threads. The integrated GPU is Broadwell GT3e, Generation 8.0. It contains 48 EUs with two slices. The second one (System 2) has an Intel Xeon E2176G CPU running at 3.7 GHz. The CPU has six cores and each core supports two threads. The integrated GPU is Coffee Lake GT2, Generation 9.5. It contains 24 EUs in a single slice. We use the DPCT in the Intel oneAPI Base Toolkit Beta8 to port CUDA codes, and the DPC++ compiler to produce binaries from the DPCT codes. Following the HIPCL installation guide, we build HIPCL from the source on each system.

The timing results are measured with the Intel OpenCL intercept layer [37]. The host timing is the total elapsed time of executing OpenCL API functions on a CPU host while the device timing the total elapsed time of executing OpenCL API functions on a GPU device. The Plugin interface (PI) is OpenCL [38], which is more mature than the new Level-ero PI. It should be mentioned that comparing performance differences of the ported codes between the two computing systems is beyond the scope of the paper. We focus on the performance implications of porting CUDA codes using HIPCL and DPCT on each system.

C. Results

While porting the applications with DPCT, intermediate configuration files are also generated for some applications. These files might be considered irrelevant in the CUDA-to-DPCT translation. In addition, the tool did not generate “cfd” codes completely, but the issue was fixed in new releases.

Table 2 and Table 3 list the host and device timing in seconds of the ported applications on the target systems, respectively. To

Table 2. Host execution time in seconds on the two systems

Host time (second)	**System 1**		**System 2**	
	HIPCL	**DPCT**	**HIPCL**	**DPCT**
miniFE	10.8	9.9	12.2	10.6
s3d	1096	73.5	1630	65.7
fft	13.9	19.2	21.2	28
md5hash	2.8	3.3	6.0	6.2
sort	8.6	28.3	10.1	17.2
lud	7.7	9.6	11.4	11.7
srad	1.58	1.58	5.3	3.8
nw	1.98	2.25	2.6	2.1
particlefilter	27.3	27	48.1	47.3
kmeans	106	112	356	346
hybridsort	1.95	1.74	2.1	1.65
hotspot3D	3.8	4.5	3.94	2.74
heartwall	8.2	8.8	14.9	12.9
gaussian	10.4	11	9.2	9.8
cfd	4.1	4.2	9.7	6.4
bfs	0.25	0.56	0.37	0.45
backprop	1.77	2.3	1.65	1.74
b+tree	0.23	0.58	0.32	0.38

Table 3. Device execution time in seconds on the two systems

Device time (second)	**System 1**		**System 2**	
	HIPCL	**DPCT**	**HIPCL**	**DPCT**
miniFE	8.9	8.8	6.5	9.44
s3d	1.07	0.83	1.5	0.92
fft	5.6	17	11.6	26.2
md5hash	2.6	2.6	5.7	5.7
sort	7.8	27.5	9.4	16.6
lud	6.3	8.45	10.5	11.2
srad	0.62	0.79	1.9	1.8
nw	0.86	0.91	1.5	1.1
particlefilter	26.4	26.3	47.3	46.8
kmeans	105	109	341	332
hybridsort	0.89	0.87	1.1	1.06
hotspot3D	3.8	3.9	5.95	4.1
heartwall	8	8.3	14.5	12.7
gaussian	10.3	10.4	11.1	11.1
cfd	3.3	3.4	8.4	5.2
bfs	0.025	0.025	0.075	0.074
backprop	1.1	1.42	1.12	1.22
b+tree	0.0039	0.0068	0.009	0.013

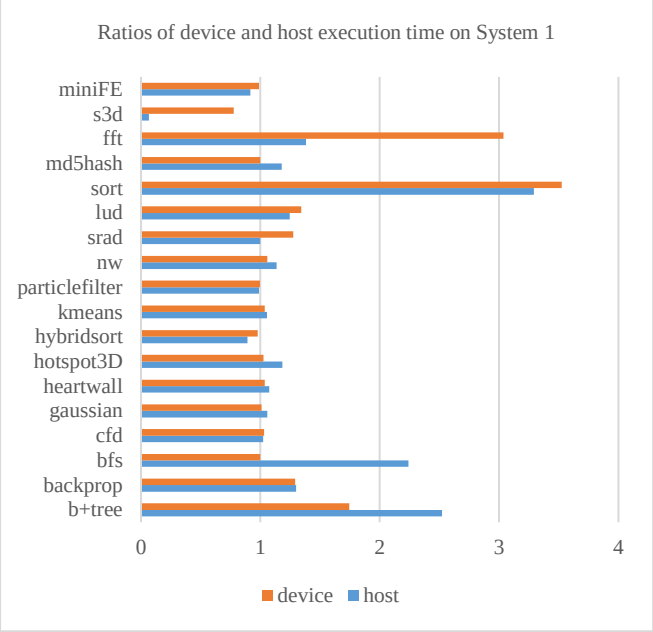

Fig 2a. The ratios of host and device execution time of the applications ported using DPCT over those using HIPCL on System 1

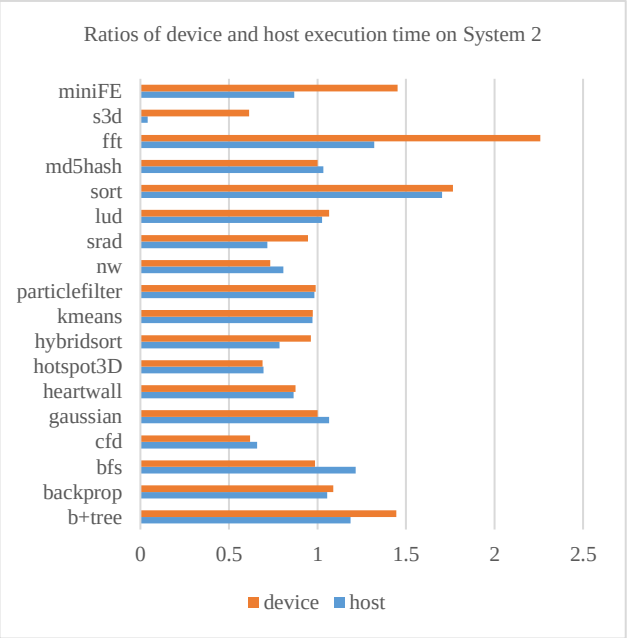

Fig 2b. The ratios of host and device execution time of the applications ported using DPCT over those using HIPCL on System 2

evaluate the performance of the ported codes, we compute the ratios of the host and device time of an application ported using DPCT over those ported using HIPCL, respectively. When the ratio is above one, the execution time of a DPCT application is longer. Figures 2 show the ratios on the two systems. While HIPCL can achieve higher or similar performance for most applications, the host execution time of "s3d" is 1096 seconds (s) and 1630 s on the two systems, respectively. Performance profiling shows that building the 27 kernels at runtime takes more than 85% of the host time. This suggests that such runtime overhead become prohibitive for large applications that contain many static kernels.

Based on the performance gaps shown in Figures 2, we will focus on the representative applications (i.e., "sort", "fft", "bfs") which perform poorly using DPCT on the two systems. Figure 2a shows that the device time of "sort" is more than three times longer. There are three static kernels in the "sort". Profiling the application shows that the third kernel, which performs a bottom scan, is a performance bottleneck. Comparing the GPU assembly of the kernels, we find that two additional instructions are generated for synchronized global fence flushing. The flushing occurs 67 times in the assembly, and the two instructions are executed 262144 times for each flushing. The flushing operations cause EUs in a GPU to stall for more than 80% of the execution time. Looking back at the DPCT kernels, we realize that the fence space of a work-group barrier is global rather than local [2]. A global fence stalls the execution of a GPU device for global memory synchronization, significantly reducing the efficiency of GPU computing when there are many synchronization points in a kernel.

It turns out that 11 out of 18 applications contain memory synchronization in their kernels. After optimizing these applications with local memory fence, we re-evaluate their performance, and find that seven applications see performance

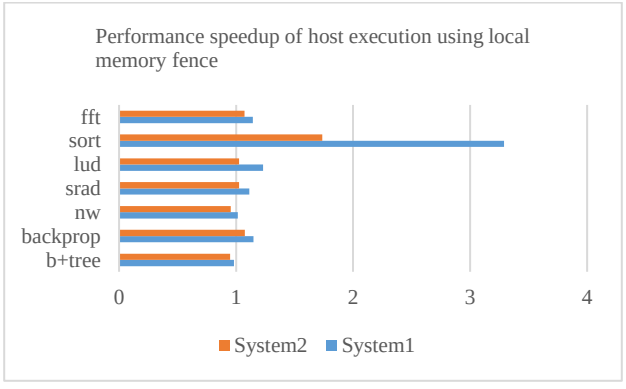

Fig 3a. Speedup of the host execution time of the applications when using local memory fence on the two systems

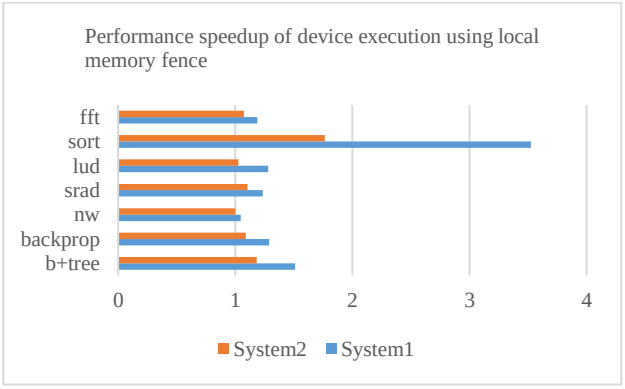

Fig 3b. Speedup of the device execution time of the applications when using local memory fence on the two systems

```
T2 data[8];
for ( int j = 1; j < 8; j++ ){ // unroll the loop
  data[j] = cmplx_mul( data[j],
  exp_i(((T)-2*(T)M_PI*reversed[j]/(T)512)*tid) );
}
…
…
for ( int j = 1; j < 8; j++ ){ // unroll the loop
  data[j] = cmplx_mul( data[j],
  exp_i(((T)-2*(T)M_PI*reversed[j]/(T)64)*hi) );
}
```

Listing 1. The loops which compute the complex values for the data array in "fft"

improvement in host and/or device execution time as shown in Figures 3. Particularly, the execution time of "sort" is now almost the same for HIPCL and DPCT. For the remaining four applications, there is no further performance improvement, indicating that memory synchronization is not on the performance critical path.

While the optimization also improves the performance of "fft", the device execution time of the application is still more than two times slower than that of the HIPCL version. FFT and inverse FFT are the two kernels in "fft". Performance profiling and the GEN assembly of the two kernels indicate that HIPCL can unroll the loops in the kernels automatically whereas they are not unrolled by the DPC++ compiler. Without loop unrolling, additional private memory is allocated by the compiler to store the 8-element array as shown in Listing 1. Hence, the compiler is unable to put the array data in the register file of an EU in a GPU for efficiency. After the loops are unrolled fully, the device execution time becomes the same for HIPCL and DPCT on each system, and the host execution time is 10.6 s on System 1 and 17.2 s on System 2, approximately 24% and 19% lower than the corresponding HIPCL time, respectively.

As shown in Figure 2a, the host execution time of "bfs" and "b+tree" ported using DPCT are approximately two times longer than those ported using HIPCL on System 1. Table 4 breaks down the execution time of the OpenCL API functions which account for 90% or more of the total host time on System 1. The results show that most of the host execution time is spent on "clBuildProgram" for the HIPCL implementation while "clCreateContext" and "clLinkProgram" consume most of the time for the DPCT implementation. "clBuildProgram" compiles and links a program executable from the program source or binary. "clLinkProgram" links compiled program objects and libraries for a specific device(s) in the OpenCL context. Reducing the link time will reduce the runtime overhead on a host. While the overhead is negligible when device execution time of an application is significantly longer, there is no benefit of offloading computation to a GPU for performance improvement when kernel computation time is less than runtime overhead.

Table 4. Breakdown of the host execution time of "bfs" on System 1

OpenCL API	**HIPCL time**	**DPCT time**
clGetPlatformIDs	0.001 s	0.05 s
clBuildProgram	0.22 s	N/A
clCreateContext	10 us	0.29 s
clLinkProgram	N/A	0.18 s
Host time	0.23 s	0.58 s

IV. Related Work

MCUDA is a source-to-source translator built upon the Cetus compiler for converting a CUDA program to a program for a multi-thread program running on a CPU [39]. MCUDA is intended to broaden the applicability of a previously accelerator-specific programming model to a CPU architecture. Swan provides a high-level library for an application to call Swan API which is then mapped to the CUDA or OpenCL API [18]. The authors point out that OpenCL lacks CUDA's C-language extensions which can simplify the host program's management of GPU code. Coriander is a compiler and runtime for running CUDA applications on OpenCL 1.2 devices [19]. The author prefers to maintain a single codebase which can run on devices from any vendor for low maintenance cost. CU2CL is a source-to-source translator built upon the Clang compiler for converting a CUDA program to an OpenCL program [20]. Contrary to the assumption that translating CUDA to OpenCL is effectively a one-to-one mapping process, translating certain parts of CUDA requires a deeper understanding of both APIs to find suitable corresponding constructs. Hence, these projects show the significance and challenges of achieving CUDA portability for CPUs and GPUs. A survey shows that DPCT has been used to convert CUDA codes in applications [12, 40, 41] and math libraries [42, 43]. However, users will need to change generated codes manually for CUDA features which are not fully supported by the tool.

V. Conclusion

In this paper, we evaluate the performance of the CUDA applications ported using HIPCL and DPCT on Intel GPUs. We find that HIPCL's runtime overhead will become prohibitive when building a large application containing many distinct kernels. HIPCL is a new library, so we expect that the potential overhead will be mitigated in the future release of the software. On the other hand, the link time in the OpenCL runtime of the Intel oneAPI toolkit may discourage a user from offloading computation to a GPU for performance improvement. Performance analysis shows that we need to manually change DPCT programs to specify the appropriate address space for memory synchronization fence. The DPC++ compiler may be improved to identify the opportunity of loop unrolling in a kernel for performance enhancement. No tools are perfect in translating a CUDA application. With the growth of the two promising toolchains for CUDA portability, we will evaluate HIPCL and DPCT using more applications in our future work.

Acknowledgment

We sincerely appreciate the reviewers for their constructive criticism and the development teams for improving HIPCL and DPCT. This research was supported by the US Department of Energy Advanced Scientific Computing Research program under Contract No. DE-AC05-00OR22725. The results presented were obtained using the Chameleon testbed and the Intel DevCloud.

REFERENCES

[1] Kirk, D., 2007, October. NVIDIA CUDA software and GPU parallel computing architecture. In ISMM (Vol. 7, pp. 103-104).

[2] Munshi, A., Jacobs, I.S., Bean, C.P., Rado, G.T. and Suhl, H., 2007. Khronos OpenCL Working Group. The OpenCL Specification, Version 1, pp.271-350.

[3] Czajkowski, T.S., Aydonat, U., Denisenko, D., Freeman, J., Kinsner, M., Neto, D., Wong, J., Yiannacouras, P. and Singh, D.P., 2012, August. From OpenCL to high-performance hardware on FPGAs. In 22nd international conference on field programmable logic and applications (FPL) (pp. 531-534). IEEE.

[4] Wong, M., Richards, A., Rovatsou, M. and Reyes, R., 2016. Khronos's OpenCL SYCL to support heterogeneous devices for C++.

[5] Ke, Y., Agung, M. and Takizawa, H., 2021, January. neoSYCL: a SYCL implementation for SX-Aurora TSUBASA. In The International Conference on High Performance Computing in Asia-Pacific Region (pp. 50-57).

[6] Jin, Z., 2020. The Rodinia Benchmark Suite in SYCL (No. ANL/ALCF-20/06). Argonne National Lab.(ANL), Argonne, IL (United States).

[7] Constantinescu, D.A., Navarro, A., Corbera, F., Fernández-Madrigal, J.A. and Asenjo, R., 2020. Efficiency and productivity for decision making on low-power heterogeneous CPU+ GPU SoCs. The Journal of Supercomputing, pp.1-22.

[8] Joó, B., Kurth, T., Clark, M.A., Kim, J., Trott, C.R., Ibanez, D., Sunderland, D. and Deslippe, J., 2019, November. Performance portability of a Wilson Dslash stencil operator mini-app using Kokkos and SYCL. In 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC) (pp. 14-25). IEEE.

[9] Jin, Z. and Finkel, H., 2019, December. A Case Study of k-means Clustering using SYCL. In 2019 IEEE International Conference on Big Data (Big Data) (pp. 4466-4471). IEEE.

[10] Jin, Z. and Finkel, H., 2019, November. Evaluation of Medical Imaging Applications using SYCL. In 2019 IEEE International Conference on Bioinformatics and Biomedicine (BIBM) (pp. 2259-2264). IEEE.

[11] Reguly, I.Z., 2019, November. Performance portability of multi-material kernels. In 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC) (pp. 26-35). IEEE.

[12] Christgau, S. and Steinke, T., 2020, May. Porting a Legacy CUDA Stencil Code to oneAPI. In 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (pp. 359-367). IEEE.

[13] Deakin, T. and McIntosh-Smith, S., 2020, April. Evaluating the performance of HPC-style SYCL applications. In Proceedings of the International Workshop on OpenCL (pp. 1-11).

[14] Burns, R., Lawson, J., McBain, D. and Soutar, D., 2019, May. Accelerated neural networks on OpenCL devices using SYCL-DNN. In Proceedings of the International Workshop on OpenCL (pp. 1-4).

[15] Goli, M., Iwanski, L. and Richards, A., 2017, May. Accelerated machine learning using TensorFlow and SYCL on OpenCL Devices. In Proceedings of the 5th International Workshop on OpenCL (pp. 1-4).

[16] Babej, M. and Jääskeläinen, P., 2020, April. HIPCL: Tool for Porting CUDA Applications to Advanced OpenCL Platforms Through HIP. In Proceedings of the International Workshop on OpenCL (pp. 1-3).

[17] https://software.intel.com/en-us/get-started-with-intel-dpcpp-compatibility-tool

[18] Harvey, M.J. and De Fabritiis, G., 2011. Swan: A tool for porting CUDA programs to OpenCL. Computer Physics Communications, 182(4), pp.1093-1099.

[19] Perkins, H., 2017, May. CUDA-on-CL: a compiler and runtime for running NVIDIA CUDA C++ 11 applications on OpenCL™ 1.2 Devices. In Proceedings of the 5th International Workshop on OpenCL (pp. 1-4).

[20] Gardner, M., Sathre, P., Feng, W.C. and Martinez, G., 2013. Characterizing the challenges and evaluating the efficacy of a CUDA-to-OpenCL translator. Parallel Computing, 39(12), pp.769-786.

[21] Sathre, Paul, Mark Gardner, and Wu-chun Feng. On the portability of CPU-accelerated applications via automated source-to-source translation. In Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, pp. 1-8. 2019.

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

[23] https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-FAQ.html#hip-faq

[24] https://github.com/ROCm-Developer-Tools/HIPIFY

[25] https://www.khronos.org/spir/

[26] Reinders, J., Ashbaugh, B., Brodman, J., Kinsner, M., Pennycook, J. and Tian, X., 2021. Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL. Springer Nature.

[27] https://github.com/zjin-lcf/oneAPI-DirectProgramming

[28] Gera, P., Kim, H., Kim, H., Hong, S., George, V. and Luk, C.K., 2018, April. Performance characterisation and simulation of intel's integrated GPU architecture. In 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (pp. 139-148). IEEE.

[29] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H. and Skadron, K., 2009, October. Rodinia: A benchmark suite for heterogeneous computing. In 2009 IEEE international symposium on workload characterization (IISWC) (pp. 44-54). IEEE.

[30] Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V. and Vetter, J.S., 2010, March. The scalable heterogeneous computing (SHOC) benchmark suite. In Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (pp. 63-74).

[31] Barrett, R.F., Tang, L. and Hu, S.X., 2014. Performance and Energy Implications for Heterogeneous Computing Systems: A MiniFE Case Study (No. SAND2014-20215). Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[32] Memeti, Suejb, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. In Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing, pp. 1-6. 2017.

[33] Mishra, A., Li, L., Kong, M., Finkel, H. and Chapman, B., 2017, November. Benchmarking and evaluating unified memory for OpenMP GPU offloading. In Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC (pp. 1-10).

[34] Zohouri, H.R., Maruyama, N., Smith, A., Matsuda, M. and Matsuoka, S., 2016, November. Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In SC16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 409-420). IEEE.

[35] Lopez, M.G., Young, J., Meredith, J.S., Roth, P.C., Horton, M. and Vetter, J.S., 2015, November. Examining recent many-core architectures and programming models using SHOC. In Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems (pp. 1-12).

[36] Fang, J., Varbanescu, A.L. and Sips, H., 2011, September. A comprehensive performance comparison of CUDA and OpenCL. In 2011 International Conference on Parallel Processing (pp. 216-225). IEEE.

[37] Ashbaugh, B., 2018, May. Debugging and Analyzing Programs Using the Intercept Layer for OpenCL Applications. In Proceedings of the International Workshop on OpenCL (pp. 1-2).

[38] https://intel.github.io/llvm-docs/PluginInterface.html

[39] Stratton, J.A., Stone, S.S. and Wen-mei, W.H., 2008, July. MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In International Workshop on Languages and Compilers for Parallel Computing (pp. 16-30). Springer, Berlin, Heidelberg.

[40] https://github.com/intel/supra-on-oneapi

[41] Phillips, J.C. and et al., 2020. Scalable molecular dynamics on CPU and GPU architectures with NAMD. The Journal of chemical physics, 153(4), p.044130.

[42] Anzt, H., Cojean, T., Chen, Y.C., Flegar, G., Göbel, F., Grützmacher, T., Nayak, P., Ribizel, T. and Tsai, Y.H., 2020. Ginkgo: A high performance numerical linear algebra library. Journal of Open Source Software, 5(52), p.2260.

[43] https://techdecoded.intel.io/resources/migrating-from-cuda-to-dpc-using-the-intel-dpc-compatibility-tool/