ORNL/TM-2021/2388

Oak Ridge National Laboratory

Zheming Jin

December 2021

OAK RIDGE National Laboratory

ORNL IS MANAGED BY UT-BATTELLE LLC FOR THE US DEPARTMENT OF ENERGY

ORNL/TM-2021/2338

Computer Science and Mathematics Division

THE RODINIA BENCHMARKS IN SYCL

Zheming Jin

December 2021

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-BATTELLE LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

The Rodinia Benchmarks in SYCL

***Abstract*—The Rodinia Benchmark Suite in SYCL was first published as a technical report at Argonne National Laboratory in June 2020. While porting more programs to SYCL, the Rodinia benchmark suites were merged into the heterogenous computing benchmarks (HeCBench). Since then, the source codes and scripts of the benchmark suite have been updated to fix compile- and run-time issues that users encountered. With the recent SYCL compilers from Codeplay and Intel, this report presents the experimental results of evaluating these benchmarks on the Intel CPU and GPU devices.**

I. INTRODUCTION

As opposed to the Open Computing Language (OpenCL) programming model in which host and device codes are generally written in two programming languages [1], SYCL can combine host and device codes for an application in a type-safe way to improve development productivity and performance portability [2].

Rodinia is a widely used benchmark suite for heterogeneous computing [3,4,5,6,7,8,9,10,11,12]. Hence, the OpenCL implementations of the benchmark suite were ported to SYCL manually. The SYCL benchmark suite is an open-source project[1] for tracking the development of the mainstream SYCL compilers [13,14,15], and for developers and researchers interested in programming productivity, performance analysis, and portability across different computing platforms [16,17,18,19,20,21,22,23,24,25,26,27].

The remainder of the report is organized as follows. Section II introduces the SYCL programming model, shows the major differences between an OpenCL program and a SYCL program, and gives a summary of the benchmark suite. In Section III, we describe our SYCL implementations in more details. Section IV evaluates the SYCL benchmarks on computing platforms with Intel central processing units (CPUs) and graphics processing units (GPUs). Section V concludes the report.

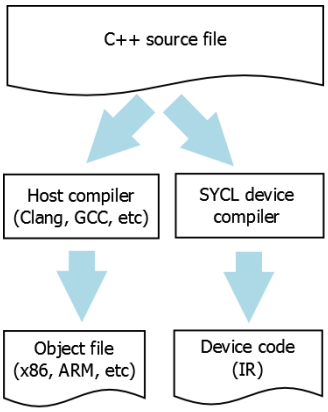

Figure 1. SYCL is a single-source programming model

II. BACKGROUND

A. SYCL

C++ AMP, CUDA, HIP, Thrust are representative single-source C/C++ programming models for accelerators [28]. Such languages can be type-checked as everything sits in a single source file. They facilitate offline compilation so that the binary can be checked at compile time. A SYCL program, based on a single-source C++ model as shown in Figure 1, can be compiled for a host while kernel(s) are extracted from the source and compiled for a device. A SYCL device compiler parses a SYCL application and generates intermediate representations (IRs). A standard host compiler parses the same application to generate native host code. The SYCL runtime will load IRs at runtime, enabling other compilers to parse it into native device code. Hence, people can continue to use existing toolchains for a host platform and choose preferred device compilers for a target platform.

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. Most of the abstraction features of C++, such as templates, classes, and operator overloading, are available for a kernel function in SYCL. A SYCL application is logically structured in three scopes: the kernel scope, the application scope, and the command-group scope. The kernel scope specifies a single-kernel function that will be executed on a device after compilation. The command-group scope specifies a unit of work that will comprise of a kernel function and buffer accessors. The application scope specifies all other codes outside of a command-group scope. A SYCL kernel function may be defined by the body of a lambda function, by a function object or by the binary generated from an OpenCL kernel string. Although an OpenCL kernel is interoperable in the SYCL programming model, we use a lambda function for each kernel in a benchmark.

TABLE I. MAPPING FROM OPENCL TO SYCL

Step	OpenCL program	SYCL program
1	Platform query	Device selector class
2	Device query of a platform	
3	Create context for devices	
4	Create command queue for context	Queue class
5	Create memory objects	Buffer class
6	Create program object	Lambda expressions
7	Build a program	
8	Create kernel(s)	
9	Set kernel arguments	
10	Enqueue a kernel object for execution	Submit a SYCL kernel to a queue
11	Transfer data from device to host	Implicit via accessors
12	Event handling	Event class
13	Release resources	Implicit via destructors

1 The Rodinia benchmarks in SYCL are available at https://github.com/zjin-lcf/HeCBench

Unified shared memory (USM) and buffers are two major ways to manage memory data in SYCL. USM is pointer-based, which is familiar to C/C++ programmers. The unified address space in USM encompasses hosts and one or more devices, reducing the barrier to porting existing C++ programs. Buffers, which are represented by the buffer template class, provides an abstract view of memory, but they are not directly accessed by a program and are instead used through accessor objects. For the Rodinia benchmarks in SYCL, the choice of memory management is SYCL buffers.

Table I lists the steps of writing an OpenCL application and their corresponding steps in SYCL. The first three steps in OpenCL are reduced to the instantiation of a device selector class in SYCL. A selector searches a device of a user's provided preference (e.g., GPU) at runtime. The SYCL queue class encapsulates a queue for scheduling kernels on a device. A kernel function in SYCL, which can be invoked as a lambda function, is grouped into a command group object, and then it is submitted to execution via command queue. Hence, steps 6 to 10 in OpenCL are mapped to the definition of a lambda expression and submission of its command group to a SYCL queue. Data transfers between a host and a device can be implicitly realized by accessor objects, and the event handling can be handled by SYCL event class. Releasing the allocated sources of queue, program, kernel, and memory objects in SYCL is handled by the SYCL runtime which implicitly calls destructors inside scopes. Compared to the number of OpenCL programming steps, the SYCL programming model cuts the number of programming steps by half with higher abstractions, reducing a developer's burden of managing OpenCL device, program, kernel, and memory objects.

B. Rodinia

Rodinia is a widely used open-source benchmark suite for heterogeneous computing. Table II lists the name in alphabetical order of each benchmark, its application domain, the number of kernels in the benchmark, and the number of kernel arguments for each kernel. Among the 21 benchmarks, the maximum number of kernels are 7 for the "hybridsort" benchmark, and the maximum number of kernel arguments are 34 for the "heartwall" benchmark. The large dataset, which are needed for some benchmarks, are not included in the GitHub repository. They can be downloaded at http://lava.cs.virginia.edu/Rodinia/download.htm.

TABLE II. SUMMARY OF THE RODINIA BENCHMARKS

Benchmark name	Application domain	Kernel counts	Argument counts for each kernel
b+tree	Search	2	10, 11
backprop	Pattern recognition	2	6, 8
bfs	Graph algorithm	1	2
cfd	Fluid dynamics	5	3, 3, 4, 5, 10
dwt2d	Video compression	3	3, 5, 7
gaussian	Linear algebra	2	5, 5
heartwall	Medical imaging	1	34
hotspot	Physics simulation	1	13
hotspot3D	Physics simulation	1	14
hybridsort	Sorting algorithm	7	3, 3, 5, 5, 5, 5, 6
kmeans	Data mining	2	4, 8
lavaMD	Chemistry	1	6
leukocyte	Medical imaging	3	7, 10, 10
lud	Linear algebra	3	4, 5, 6
myocyte	Biological simulation	1	5
nn	Data mining	1	5
nw	Bioinformatics	2	12, 12
particlefilter	Medical imaging	4	2, 6, 8, 20
pathfinder	Grid traversal	1	12
srad	Image processing	6	2, 2, 3, 4, 14, 14
streamcluster	Data mining	2	3, 10

III. IMPLEMENTATIONS

In consideration of the rapidly evolving SYCL programming model [29], we would like to summarize the SYCL features utilized in the benchmarks.

A. Buffer Construction

In SYCL, a host application uses instances of the SYCL buffer class to allocate memory in global, local, and constant address spaces. A SYCL buffer can handle both storage and ownership of data. In addition, a buffer is destroyed when it goes out of scope.

Table III lists the ways a buffer can be constructed and its initial values after construction. The destruction behavior indicates if the SYCL runtime will block until all work in queues on the buffer have completed. For the benchmark suite, we use the first two methods for constructing buffers.

TABLE III. SUMMARY OF SYCL BUFFER MANAGEMENT

Construction method	Initial buffer content after construction	Destruction behavior
Buffer size	Unspecified	Non-blocking
Associated host memory	Contents of host memory	Blocking
Unique pointer to host data	Contents of host data	Blocking
Shared pointer to host data	Contents of host data	Blocking
A pair of iterator values	Data from the range defined by the iterator pair	Non-blocking
Container	Contents of the container	Blocking (1)

(1) Please see the buffer synchronization rules in the SYCL specification for more details

B. Buffer Access Mode

SYCL accessors allow a user to specify the types (e.g., global memory or constant memory) of data access, and the SYCL implementation ensures that the data is accessed appropriately. A device accessor, which is the default access type, allows a kernel to access data on a device. In contrast, a host accessor gives access to data on a host. A device accessor can only be constructed within command groups whereas a host accessor can be created outside command groups. Constructing a host accessor is blocking by waiting for all previous operations on the underlying buffer to complete. When accessing the contents of a device buffer before the buffer is destroyed in a host program, we could use a host accessor to access the contents managed by the device buffer.

An accessor must be specified with an access mode shown in Table IV. Discarding write indicates that previous contents of a device buffer is not preserved, which implies

TABLE IV. SYCL BUFFER ACCESS MODES

Access mode	Description
Read	Read-only access to the content of a buffer
Write	Write-only access to the content of a buffer
Read/Write	Read and write access to a buffer
Discard Write	Write-only access to the content of a buffer. Discard any previous contents of the data the accessor refers to
Discard Read/Write	Read and write access to the content of a buffer. Discard any previous contents of the data the accessor refers to
Atomic	Atomic access to the content of a buffer

that it is not necessary to copy data from a host to a device before the buffer is accessed. It is important to specify the access mode correctly; otherwise, SYCL compilers will report an error when a kernel function tries to write to a read-only buffer. On the other hand, a read-only accessor to a buffer disables data copy from a device to a host when the buffer is destroyed.

C. Data Movement between Host and Device

For the OpenCL implementations of the benchmark suite, data transfers between a host and a device are explicitly made with the OpenCL built-in functions "clEnqueueReadBuffer()" and "clEnqueueWriteBuffer()". In the SYCL implementations, we rely on implicit and/or explicit data transfers. When a buffer is constructed with associated host memory as shown in Table III, the SYCL runtime will copy data from a host to a device before a kernel is launched, and optionally copy data back from a device to a host before the buffer is destroyed. Without explicit data copy specified in a SYCL program, a SYCL compiler may generate OpenCL built-in functions "clEnqueueMapBuffer()" and "clEnqueueUnmapMemObject()" for mapping data between a host and a device. On the other hand, data copy from a device to a host can be disabled explicitly using the method "set_final_data(nullptr)" of the SYCL buffer class.

For explicit data transfers, we use the copy method of the command group handler. An explicit copy operation requires the specifications of a source and a destination. When an accessor is the source of the operation, the destination can be a host pointer or another accessor. When an accessor is the destination of the explicit copy operation, the source can be a host pointer or another accessor.

D. Kernel Execution Order

In OpenCL, a command queue is required to transfer data between a host and a device, and to ensure different kernels execute in the correct order. In contrast, SYCL provides an abstraction that only requires users to specify which data are needed to execute a kernel. By specifying access modes and types of memory for each kernel, a directed acyclic dependency graph for different kernels is constructed at runtime based on the relative order of command-groups submissions to a queue. The runtime will guarantee that kernels are executed in an order that guarantees correctness. By default, SYCL queues execute kernel functions in an out-of-order fashion. An in-order queue, which is an extension to the default queue property [30], is not used in our implementations.

E. Kernel Execution Model

Conceptually, the SYCL kernel execution model is equivalent to the OpenCL kernel execution model. SYCL supports an N-dimensional ($N \leq 3$) index space, and the space is represented via the "nd_range<N>" class. Each work-item in the space can be identified by the type "nd_item<N>". The type encapsulates a global identifier (ID), a local ID, a work-group ID, synchronization operations, etc.

A SYCL runtime creates a SYCL handler object to define and invoke a SYCL kernel function in a command group. A kernel can be invoked as a single task, a basic data-parallel kernel, an OpenCL-style kernel, or a hierarchical parallel kernel. In our experiment, we invoke a variant of the "parallel_for" member function that enables low-level functionality of work-items and work-groups for a data-parallel kernel. The variation allows us to specify both global and local ranges, perform the synchronization of work-items in each cooperating work-group, and create accessors to local memory. These functions are helpful for the smooth migration of an OpenCL kernel to a SYCL kernel.

IV. EXPERIMENT

A. Setup

The benchmarks are compiled with the two commercial SYCL compilers, the Intel DPC++ compiler 2021.4.0 and the Community Edition of the Codeplay ComputeCpp compiler 2.7.0. Two computing servers are used for evaluating the benchmarks. The first server contains an Intel Xeon E-2176G CPU running at 3.7 GHz. The CPU has six cores, and each core supports two threads. The integrated

TABLE V. PROBLEM SIZE FOR EACH BENCHMARK IN RODINIA

Benchmark	Problem size
b+tree	1 million keys, 10000 bundled queries, a range search of 6000 bundled queries with the range of each search 3000
backprop	65536 input nodes
bfs	1 million vertices
cfd	97K elements
dwt2d	1024×1024 image, forward 5/3 transform
gaussian	4096×4096 matrix
heartwall	104 frames
hotspot	512×512 data points
hotspot3D	512×512 data points
hybridsort	50000000 elements
kmeans	494020 points, 34 features
lavaMD	1000 boxes
leukocyte	10 frames
lud	8192×8192 data points
myocyte	100 timesteps
nn	5 nearest neighbors
nw	16384×16384 data points
particlefilter	400000 points
pathfinder	100000×100 2D grid
srad	512×512 data points
streamcluster	65536 points 256 dimensions

GPU (UHD Graphics 630) is Coffee Lake GT2, Generation 9.5. The second server has an Intel Xeon E3-1585 v5 CPU running at 3.5 GHz. The CPU has four cores, and each core supports two threads. The integrated GPU (Iris Pro Graphics 580) is Skylake GT4e, Generation 9.0. It contains 72 execution units.

For the GPU compute runtime, the device version is OpenCL 3.0 NEO on both platforms. The driver versions are 21.33.20678 and 21.21.0 on the two platforms, respectively. For the CPU runtime, the device version is OpenCL 3.0, and the driver version is 2021.12.9.0.24_005321 on both platforms. The maximum work-group size is 256 and 8192 on a GPU and a CPU, respectively. The operating systems are Ubuntu 20.04 and OpenSUSE 15.3, respectively. The compiler options are "-O3 -no-serial-memop -sycl-driver" for the ComputeCpp compiler, and "-O3" for the Intel DPC++ compiler.

The problem sizes for the benchmarks are listed in Table V. For each benchmark, we use the same work-group size for the CPU and GPU unless different work-group sizes are specified in the benchmark. While adjusting the problem size and turning the work-group size may further improve the raw performance of SYCL kernels on a target platform, we are more concerned with developing SYCL programs to support the development of the SYCL compilers.

B. Experimental Results

We execute each benchmark four times and measure the device timing reported by the Intel OpenCL profiler [31]. The device timing is the total elapsed time of executing kernel(s) on a device specified by the SYCL's device selector. The host timing, which is the total elapsed time of executing OpenCL runtime functions on a CPU host, is not considered for performance evaluation. We find that the host timing of a benchmark compiled with the ComputeCpp compiler does not necessarily include the device timing, whereas device timing is part of host timing for a benchmark compiled with the DPC++ compiler. The discrepancy of host timing between the two compilers is related to whether certain OpenCL runtime functions invoked on a host wait for the completion of kernel execution on a device.

Tables VI and VII list the CPU and GPU device execution time in seconds of the benchmarks on the two computing platforms, respectively. "DPC++" and "CPC++" refer to the Intel DPC++ compiler and the Codeplay ComputeCpp compiler, respectively. The timing results of the "hybridsort" benchmark are not available (N/A) except the GPU timing of the benchmark compiled with DPC++.

Figures 2 and 3 compare the CPU and GPU device timing for each benchmark using the two SYCL compilers.

TABLE VI. CPU and GPU device execution time on the Intel Xeon E-2176G and UHD Graphics 630, respectively

Time (seconds)	**DPC++ GPU device**	**CPC++ GPU device**	**DPC++ CPU device**	**CPC++ CPU device**
b+tree	0.0104	0.0077	0.006	0.012
backprop	0.0045	0.004	0.002	0.003
bfs	0.0733	0.04	0.0251	0.023
cfd	3.0	4.0	4.1	11.2
dwt2d	0.044	0.028	0.009	0.015
gaussian	69.1	68.4	15.7	30.9
heartwall	15.6	14.8	12.4	23
hotspot	0.1	0.044	0.054	0.11
hotspot3D	4.1	4.2	5.8	6.5
hybridsort	1.24	N/A	N/A	N/A
kmeans	120.5	115.8	78.9	177.2
lavaMD	1.3	1.5	4.86	5.36
leukocyte	4.86	5.6	3.9	21.7
lud	10.7	11.36	5.5	13.8
myocyte	0.56	1.47	0.27	0.26
nn	0.15	0.248	0.676	0.83
nw	0.71	0.84	0.58	0.97
particlefilter	49.5	45.5	25.3	27.4
pathfinder	3.36	2.94	28.9	21.4
srad	2.1	1.2	1.1	2.2
streamcluster	7.1	9.5	6.1	12.4

TABLE VII. CPU and GPU device execution time on the Intel Xeon E3-1585 v5 and Iris Pro Graphics 750, respectively

Time (seconds)	**DPC++ GPU device**	**CPC++ GPU device**	**DPC++ CPU device**	**CPC++ CPU device**
b+tree	0.0014	0.0029	0.009	0.017
backprop	0.00089	0.001	0.0024	0.004
bfs	0.0167	0.018	0.034	0.027
cfd	2.02	1.98	5.1	17.7
dwt2d	0.0069	0.006	0.011	0.019
gaussian	63.6	53.6	20.5	52.3
heartwall	3.48	3.46	18.8	36.3
hotspot	0.015	0.016	0.058	0.175
hotspot3D	1.98	1.93	6.18	10
hybridsort	0.74	N/A	N/A	N/A
kmeans	69.4	69.7	124.3	280.7
lavaMD	0.44	0.53	7.87	8.75
leukocyte	1.38	1.52	6.08	33.4
lud	4.16	4.35	7.05	22.1
myocyte	0.63	0.76	0.028	0.0282
nn	0.093	0.1	0.31	0.465
nw	0.49	0.6	0.54	1.11
particlefilter	19.6	18.3	0.39	0.44
pathfinder	1.2	1.07	40.1	34.6
srad	0.412	0.395	1.06	3.1
streamcluster	3.21	3.28	6.4	35.5

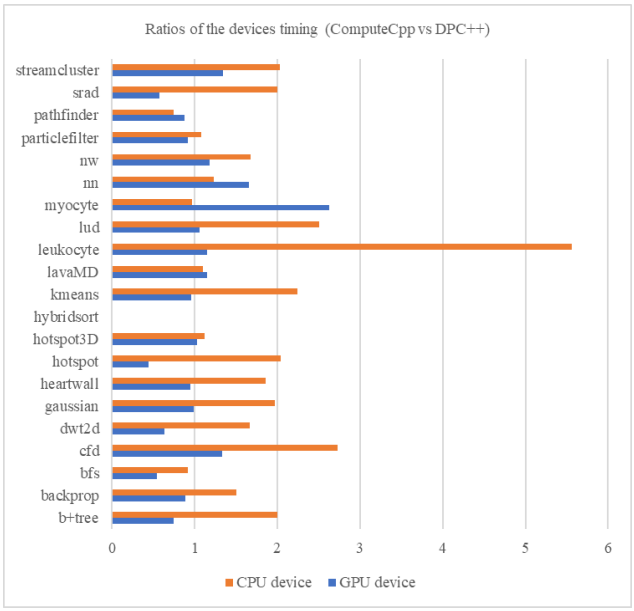

Figure 2. The ratio of CPU (Intel Xeon E-2176G) and GPU (UHD Graphics 630) device timing for each benchmark compiled with the ComputeCpp and DPC++ compilers. When the ratio is over one for a benchmark, the device time is longer for the benchmark compiled with the ComputeCpp compiler.

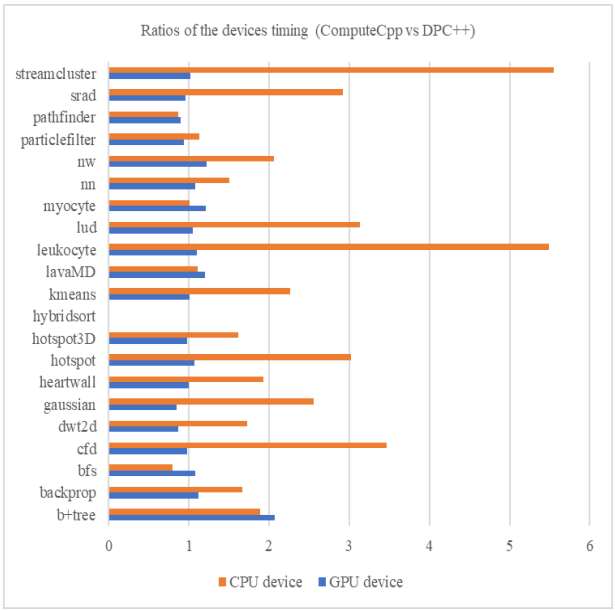

Figure 3. The ratio of CPU (Intel Xeon E3-1585 v5) and GPU (Iris Pro Graphics 750) device timing for each benchmark compiled with the ComputeCpp and DPC++ compilers. When the ratio is over one for a benchmark, the device time is longer for the benchmark compiled with the ComputeCpp compiler.

When the ratio of device timing (CPU or GPU) for a benchmark is over 1, the device time of the benchmark compiled with CPC++ is longer than that of the benchmark compiled with DPC++. Excluding the "hybridsort" benchmark whose results are mostly not available, there are 20 benchmarks in the suite. The averaged differences of the GPU device time of the benchmarks compiled with the two compilers are 5% and 8% on the two computing platforms, respectively. However, the CPU device time is on average 1.84X and 2.28X longer using CPC++ on the two computing platforms, respectively. Hence, the performance gaps of the SYCL benchmarks executing on the CPUs are much more significant than those on the GPUs.

We try to have a better understanding of the performance gaps on a CPU by profiling "leukocyte", "kmeans", "gaussian", "cfd", "lud", "hotspot", and "srad" with the Intel VTune Profiler 2021.8.0. The device time of each selected benchmark is at least 2X longer with CPC++ on the two platforms. The hotspots of the "leukocyte" benchmark are the spin time incurred by the implementation of the loop scheduler in the proprietary ComputeCpp library and the compute time of the math function "atanf" called in the "IMGVF" kernel. The single-precision floating-pointer operations in the kernel may be fully packed (512-bit) with DPC++. Similarly, the hotspots of the "kmeans" benchmark are the spin time and the minimum distance compute kernel which performs scalar single-precision floating-point operations with CPC++. Almost 100% of the single-precision floating-point operations of the "fan2" kernel in the "gaussian" benchmark can be packed to 128- or 512-bit operations with DPC++, but less than 15% of the operations can be packed to 512 bits with CPC++. For the "cfd" and "lud" benchmark, the hotspots are the "compute_flux" kernel and the "internal" kernel, respectively. Both kernels can be better optimized with DPC++ to achieve higher single-precision floating-point operations per second. For the "hotspot" and "srad" benchmarks, the spin time incurred by the implementation of the loop scheduler in the ComputeCpp library is significant in the device time on the CPU. The profiling results indicate that the major causes of the performance gaps on the CPUs are the percentage of packed (vectorized) floating-point operations that can be achieved by the compilers and the efficiency of scheduling a loop across threads at runtime.

V. SUMMARY

SYCL is becoming a promising programing model for heterogeneous computing. We apply the SYCL programming model to the widely used Rodinia benchmark suite, describe the transformations from the OpenCL implementations of the benchmarks to the SYCL implementations, and evaluate the benchmarks with the Intel DPC++ and Codeplay ComputeCpp compilers on microprocessors with a CPU and an integrated GPU. The

publicly available implementations of the benchmark suite in SYCL are important for the development of the SYCL compilers and the performance and portability studies for heterogeneous computing systems.

ACKNOWLEDGMENT

The author would like to acknowledge people at the Advanced Computing Systems Research section in Oak Ridge National Laboratory for their generous support. The author would also like to acknowledge the community for their active development of SYCL compilers. This research used resources of the Argonne Leadership Computing Facility and the Intel DevCloud. This research was supported 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.

REFERENCES

[1] Stone, J.E., Gohara, D. and Shi, G., 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering, 12(3), pp.66-73.

[2] Doumoulakis, A., Keryell, R. and O'Brien, K., 2017, May. SYCL C++ and OpenCL interoperability experimentation with triSYCL. In Proceedings of the 5th International Workshop on OpenCL (pp. 1-8).

[3] 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.

[4] Che, S., Sheaffer, J.W., Boyer, M., Szafaryn, L.G., Wang, L. and Skadron, K., 2010, December. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In IEEE International Symposium on Workload Characterization (IISWC'10) (pp. 1-11). IEEE.

[5] Wen, H. and Zhang, W., 2019, September. Improving Parallelism of Breadth First Search (BFS) Algorithm for Accelerated Performance on GPUs. In 2019 IEEE High Performance Extreme Computing Conference (HPEC) (pp. 1-7). IEEE.

[6] Memeti, S., Li, L., Pllana, S., Kołodziej, J. and Kessler, C., 2017, July. 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).

[7] Konstantinidis, E. and Cotronis, Y., 2017. A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. Journal of Parallel and Distributed Computing, 107, pp.37-56.

[8] Che, S. and Skadron, K., 2014. BenchFriend: Correlating the performance of GPU benchmarks. The International journal of high performance computing applications, 28(2), pp.238-250.

[9] 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 SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 409-420). IEEE.

[10] Landaverde, R., Zhang, T., Coskun, A.K. and Herbordt, M., 2014, September. An investigation of unified memory access performance in CUDA. In 2014 IEEE High Performance Extreme Computing Conference (HPEC) (pp. 1-6). IEEE.

[11] Shen, J., Fang, J., Sips, H. and Varbanescu, A.L., 2012, September. Performance gaps between OpenMP and OpenCL for multi-core CPUs. In 2012 41st International Conference on Parallel Processing Workshops (pp. 116-125). IEEE.

[12] Shen, J., Fang, J., Sips, H. and Varbanescu, A.L., 2013. An application-centric evaluation of OpenCL on multi-core CPUs. Parallel Computing, 39(12), pp.834-850.

[13] https://github.com/intel/llvm

[14] https://www.oneapi.com/

[15] https://www.codeplay.com/products/computesuite/computecpp

[16] 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).

[17] Aktemur, B., Metzger, M., Saiapova, N. and Strasuns, M., 2020, April. Debugging SYCL Programs on Heterogeneous Intel® Architectures. In Proceedings of the International Workshop on OpenCL (pp. 1-10).

[18] Alpay, A. and Heuveline, V., 2020, April. SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL. In Proceedings of the International Workshop on OpenCL (pp. 1-1).

[19] 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.

[20] 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.

[21] Jin, Z., 2019. Improving the Performance of Medical Imaging Applications using SYCL (No. ANL/ALCF-19/4). Argonne National Lab.(ANL), Argonne, IL (United States).

[22] 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.

[23] Deakin, T., McIntosh-Smith, S., Price, J., Poenaru, A., Atkinson, P., Popa, C. and Salmon, J., 2019, November. Performance Portability across Diverse Computer Architectures. In 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC) (pp. 1-13). IEEE.

[24] Thoman, P., Salzmann, P., Cosenza, B. and Fahringer, T., 2019, August. Celerity: High-Level C++ for Accelerator Clusters. In European Conference on Parallel Processing (pp. 291-303). Springer, Cham.

[25] Burke, T.P., 2019. Parallelization of a Proxy Transport App Using ComputeCPP and SYCL (No. LA-UR-19-25636). Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

[26] Afzal, A., Schmitt, C., Alhaddad, S., Grynko, Y., Teich, J., Forstner, J. and Hannig, F., 2018, July. Solving Maxwell's Equations with Modern C++ and SYCL: A Case Study. In 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP) (pp. 1-8). IEEE.

[27] Da Silva, H.C., Pisani, F. and Borin, E., 2016, October. A comparative study of SYCL, OpenCL, and OpenMP. In 2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW) (pp. 61-66). IEEE.

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

[29] https://www.khronos.org/sycl/

[30] https://github.com/intel/llvm/tree/sycl/sycl/doc/extensions/USM

[31] https://github.com/intel/llvm/tree/sycl/sycl/doc/extensions/OrderedQueue