ANL/ALCF-20/6

Argonne NATIONAL LABORATORY

The Rodinia Benchmark Suite in SYCL

Leadership Computing Facility

ANL/ALCF-20/6

The Rodinia Benchmark Suite in SYCL

prepared by
Zheming Jin
Leadership Computing Facility, Argonne National Laboratory

June 1, 2020

The Rodinia Benchmark Suite in SYCL

I. INTRODUCTION

As opposed to the Open Computing Language (OpenCL) programming model in which host and device codes are generally written in different 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, we port the OpenCL implementations of the benchmark suite 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].

We organize the remainder of the report 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. We evaluate the SYCL benchmarks on the Intel® central processing units (CPUs) and graphics processing units (GPUs) in Section IV. Section V concludes the report.

II. BACKGROUND

A. SYCL

C++ AMP, CUDA, HIP, Thrust C++ are representative single-source 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 (IR). A standard host compiler parses the same application to generate native host code. The SYCL runtime will load IR 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.

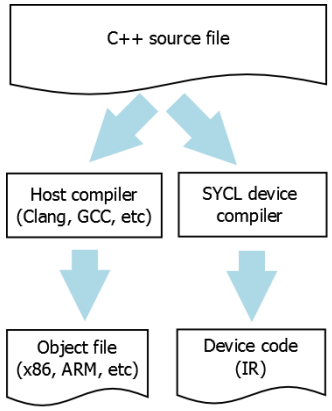

Figure 1. SYCL is a single-source programming model

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: kernel scope, application scope, and 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 lists the steps of creating 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

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 SYCL implementation of the Rodinia benchmark suite is available at https://github.com/zjin-lcf/Rodinia_SYCL

command queue. Hence, steps 6 to 10 in OpenCL are mapped to the definition of a lambda function and submission of its command group to a SYCL queue. Data transfers between a host and a device can be implicitly realized by SYCL accessors, 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 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 OpenCL kernels, and the number of kernel arguments for each kernel. Among all the benchmarks, the maximum number of kernels is 7 for the hybridsort benchmark, and the maximum number of kernel arguments is 34 for the heartwall benchmark. We carefully and manually port all the OpenCL benchmarks which are available in Rodinia to SYCL.

TABLE II. SUMMARY OF THE BENCHMARKS IN RODINIA

Benchmark	Application domain	OpenCL Kernels	OpenCL kernel arguments
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 language features utilized in our implementations and other features which are left as future work.

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	Buffer Content after Construction	Destruction Behavior
Buffer size	Uninitialized	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
OpenCL memory object	OpenCL memory object	Blocking

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 the 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 we need to access the contents of a device buffer before the buffer is destroyed in a host program, we should use a host accessor to access the memory managed by a 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 that it is not necessary to copy data from host to device before the buffer is accessed. It is important to specify the access mode correctly; otherwise, the compiler 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 to host memory when the buffer is destroyed. The access modes for the benchmark suite are Read, Write, Read/Write, and Atomic.

TABLE IV. SYCL BUFFER ACCESS MODES

Access Mode	Description
Read	Read-only access to a buffer
Write	Write-only access to a buffer
Read_write	Read and write access to a buffer
Discard_write	Write-only access to a buffer. Discard any previous contents of the data the accessor refers to
Discard_read_write	Read and write access to a buffer. Discard any previous contents of the data the accessor refers to
Atomic	Atomic access to a buffer

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 functions "clEnqueueReadBuffer()"

and "clEnqueueWriteBuffer()". In the SYCL implementations, we rely on implicit and/or explicit data transfers. Unified shared memory (USM), an extension to pointer-based programming in SYCL [30], is not used in our implementations. The unified address space encompasses the host and one or more devices, reducing the barrier to integrate SYCL code into existing C++ programs.

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 device to 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, copyback from a device to a host can be disabled using the SYCL method "set_final_data(nullptr)".

For explicit (manual) data transfers, we use the copy method of the command group handler. The explicit copy operations have 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, a directed acyclic dependency graph of different kernels is constructed at runtime based on the relative order of command-groups submissions to a queue. Queues in SYCL are out-of-order by default. An in-order queue, which is an extension to the default queue property [31], 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 is identified by the type "nd_item<N>". The type encapsulates a global identifier (ID), a local ID, a work-group ID, synchronization operations, etc.

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, enabling the smooth migration of an OpenCL kernel to a SYCL kernel.

TABLE V. SUMMARY OF THE GPUS USED IN THE EXPERIMENT

Parameter	Iris™ Pro Graphics P6300	UHD Graphics 630
Generation	Gen8	Gen9.5
Technology	14 nm	14 nm
Base/Max Freq	0.3/1.15 GHz	0.35/1.2 GHz
Embedded DRAM	128 MB	N/A
Slices/Subslices	2/6	1/3
EUs (total)	48	24
Peak single-precision GFLOPS	883	441

IV. EXPERIMENT

A. Setup

We build the applications with the Intel oneAPI Toolkit (Beta06) and Codeplay ComputeCpp community edition (version 2.0.0). We choose two computing platforms in our experiment. The first one 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 execution units (EUs)

TABLE VI. 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 images, forward 5/3 transform
gaussian	1024×1024 matrix
heartwall	104 frames
hotspot	512×512 data points
hotspot3D	512×512 data points
hybridsort	100000 elements
kmeans	494020 points, 34 features
lavaMD	1000 boxes
leukocyte	10 frames
lud	2048×2048 data points
myocyte	100 time steps
nn	5 nearest neighbors
nw	2048×2048 data points
particlefilter	400000 points
pathfinder	100000×100 2D grid
srad	512×512 data points
streamcluster	65536 points 256 dimensions

with two slices. The second one 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. A few specifications of the GPUs are listed in Table V.

For the GPU compute runtime, the device version is OpenCL 2.1 NEO and the driver version 20.12.16259. For the CPU runtime, the device version is OpenCL 2.1 and the driver version 2020.10.4.0.15. The maximum work-group size is 256 and 8192 on a GPU and a CPU, respectively. The operating system is Ubuntu 18.04. The compiler options are "-O3 -no-serial-memop -sycl-driver" for the Codeplay ComputeCpp compiler, and "-O3" for the Intel DPC++ compiler. Both compilers invoke the default GNU compiler, version 7.4.0, installed on the target platforms to compile a host program.

The problem sizes for the benchmark suite are listed in Table VI. 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 performance efficiency and raw performance of each benchmark on a target platform, we are more concerned with developing SYCL benchmarks to support the development of the SYCL compilers in terms of functionality and performance.

B. Experimental Results

We run each benchmark 10 times. For each benchmark, the average execution time includes not only kernel execution time but also communication cost and SYCL runtime overhead such as buffer construction and destruction. We argue that runtime overhead may become significant compared to kernel execution time for certain problem sizes. Table VII and Table VIII show the execution time of the benchmarks on the two computing platforms, respectively. The experimental results call for the improvement of the SYCL compilers in terms of functionality and performance.

TABLE VII. Execution time on the Intel Xeon E3-1284L v4 (CPU) and Iris Pro Graphics P6300 (GPU)

Time (seconds)	**DPC++ GPU**	**ComputeCpp GPU**	**DPC++ CPU**	**ComputeCpp CPU**
b+tree-findK	1.82	0.1	N/A	0.27
b+tree-findRangeK	1.78	0.069	N/A	0.19
backprop	0.2	0.11	0.46	0.51
bfs	0.22	0.11	0.41	0.27
cfd	4.54	4.8	13.9	18.6
dwt2d	0.6	0.31	1.89	0.97
gaussian	0.38	0.33	0.55	0.66
heartwall	N/A	8.8	N/A	29
hotspot	0.23	0.11	0.45	0.52
hotspot3D	0.36	0.3	0.68	0.82
hybridsort	0.58	N/A	N/A	N/A
kmeans	0.78	0.55	1.24	1.25
lavaMD	0.27	0.14	1.18	0.76
leukocyte-GICOV	0.346	0.182	1.12	0.37
leukocyte-dilation	0.0048	0.0051	0.0036	0.0031
leukocyte-MGVF	0.019	0.032	0.309	0.145
lud	1.2	1.13	2.2	1.36
myocyte	2.85	2.47	2.1	0.9
nn	0.16	0.1	0.27	0.2
nw	0.43	0.31	0.81	0.8
particlefilter	N/A	N/A	30.8	1.14
pathfinder	0.23	0.14	2.64	1.15
srad	0.38	0.21	1.43	0.91
streamcluster	6.2	5.7	16.1	6.9

TABLE VIII. Execution time on the Intel Xeon E2176G (CPU) and UHD Graphics 630 (GPU)

Time (seconds)	DPC++ GPU	ComputeCpp GPU	DPC++ CPU	ComputeCpp CPU
b+tree-findK	0.76	0.17	N/A	0.27
b+tree-findRangeK	0.5	0.06	N/A	0.19
backprop	0.26	0.18	0.51	0.55
bfs	0.27	0.2	0.43	0.29
cfd	4.8	7.5	7.9	12
dwt2d	0.59	0.34	1.81	0.95
gaussian	0.43	0.7	0.55	0.6
heartwall	N/A	8.3	N/A	16.2
hotspot	0.28	0.17	0.5	0.56
hotspot3D	0.46	0.41	0.59	0.71
hybridsort	0.62	N/A	N/A	N/A
kmeans	0.93	0.76	1.1	1.22
lavaMD	0.31	0.22	0.92	0.61
leukocyte-GICOV	0.398	0.24	1.13	0.41
leukocyte-dilation	0.005	0.009	0.027	0.018
leukocyte-MGVF	0.037	0.05	0.178	0.1
lud	1.32	1.3	1.9	1.1
myocyte	7.8	8.7	2.2	1.1
nn	0.21	0.16	0.35	0.26
nw	0.43	0.37	0.81	0.83
particlefilter	54.7	51	N/A	0.9
pathfinder	0.33	0.27	2.3	0.94
srad	0.64	0.58	1.35	0.9
streamcluster	10.7	11.1	11.5	8.1

V. SUMMARY

We apply the SYCL programming model to the Rodinia benchmark suite, describe the transformations from the OpenCL implementations to the SYCL implementations, and evaluate the benchmarks on microprocessors with a CPU and an integrated GPU. The publicly available implementations of the benchmark suite will track the development of the SYCL compilers, and provide programs for the study of heterogeneous systems.

ACKNOWLEDGMENT

Results presented were obtained using the Chameleon testbed supported by the National Science Foundation, and the Intel® DevCloud. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

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

Leadership Computing Facility
Argonne National Laboratory
9700 South Cass Avenue, Bldg. 240
Argonne, IL 60439

www.anl.gov

Argonne National Laboratory is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC