ORNL/TM-2022/2433

Oak Ridge National Laboratory

Zheming Jin

April 2022

OAK RIDGE
National Laboratory

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

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ORNL/TM-2022/2433

Computer Science and Mathematics Division

EXPERIENCE OF MIGRATING A PARALLEL GRAPH COLORING PROGRAM FROM CUDA TO SYCL

Zheming Jin

April 2022

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

Experience of Migrating a Parallel Graph Coloring Program from CUDA to SYCL

Abstract. We describe the experience of converting a CUDA implementation of a parallel graph coloring algorithm to SYCL. The goals are for our work to be useful to application and compiler developers by providing a detailed description of migration paths between CUDA and SYCL. We will describe how CUDA functions are mapped to SYCL functions. Evaluating the CUDA and SYCL implementations of the algorithm shows that the performance of SYCL and CUDA kernels are comparable over the test graph set on NVIDIA P100 and V100 GPUs. The SYCL program also allows for performance evaluation with the OpenCL and Level Zero interfaces and power profiling on an Intel GPU computing platform.

1 INTRODUCTION

CUDA has successfully enabled the use of a graphics processing unit (GPU) as a programmable general-purpose computing device [1]. However, CUDA is a proprietary programming model for NVIDIA GPUs. In contrast, OpenCL 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 to GPUs and other heterogeneous computing devices [2, 3]. While an OpenCL program can be compiled and executed on a variety of platforms, porting a CUDA program to OpenCL tends to be error-prone and time-consuming [4, 5]. Portability is a key objective for SYCL, a specification which defines a single-source C++ programming layer on top of OpenCL [6]. SYCL is a promising programming model for heterogeneous computing because it builds on the underlying concepts, portability, and efficiency of OpenCL while adding much of the ease of use and flexibility of single-source C++ [7].

In this work, we describe the experience of migrating a parallel implementation of a graph coloring algorithm from CUDA to SYCL. Specifically, we choose a highly optimized implementation of the algorithm in CUDA, map the CUDA program to SYCL manually, and evaluate the performance of the compute kernels on GPUs. While the example is specific to graph coloring, the experience may be valuable to application and compiler developers for the development of the SYCL programming model.

We sum up the findings of our study as follows: 1) While both CUDA and SYCL are extensions to the C and C++ programming languages, certain CUDA device property, math function, and warp primitive are not fully supported by SYCL built-in functions yet. Hence, developers will need to implement these functions based on the CUDA programming guide and the SYCL specification. 2) Most CUDA warp-level primitives, which are provided for performance optimization using explicit warp-level programming, could be mapped directly to the SYCL group functions except that a bit mask is not supported by the group functions. 3) While groups of threads known as warps have a fixed warp size of 32 in CUDA, the size must be explicitly specified using a SYCL kernel attribute to inform the SYCL runtime. 4) The CUDA and SYCL kernels are comparable in terms of kernel execution time for the test graph set on NVIDIA P100 and V100 GPUs though certain architecture-specific features in CUDA are not available in SYCL. 5) Migrating the CUDA program to SYCL allows us to evaluate the application performance with the Intel OpenCL and Level Zero compute interfaces on an Intel GPU. The OpenCL interface is on average 10.5% faster than the Level Zero interface on an Intel UHD Graphics 630 for the test graph set. The SYCL program is available at https://github.com/zjin-lcf/HeCBench/blob/master/gc-sycl.

CUDA is a mature programming model. Therefore, certain CUDA features are not yet supported in SYCL or other heterogeneous programming models [8]. Along the course of migrating CUDA programs in academia, industry, and computing facilities for functional and performance portability, we hope that our study will be useful for the SYCL communities. We have described the motivation and scope of our study in this section. Section 2 introduces briefly graph coloring and the SYCL branch of the Intel LLVM repository. Section 3 describes the migration of the graph coloring example from CUDA to SYCL. Section 4 presents the experimental results on the GPUs. Section 5 discusses related work, and Section 6 is a summary of the study.

2 BACKGROUND

2.1 Graph coloring

Graph coloring is widely used in many domains [9, 10, 11, 12, 13]. It assigns colors to all vertices of a graph such that no adjacent vertices have the same color. It is also an optimization problem of coloring a graph with minimum number of colors. It is NP-hard, so there is no known polynomial time algorithm that can solve it optimally [14]. Heuristic algorithms can color a graph with no adjacent vertices assigned the same color, but they may require more colors than the optimal algorithm [15, 16, 17, 18, 19].

In this paper, we choose a parallel implementation of a state-of-the-art graph coloring algorithm for our study. The algorithm increases parallelism by a factor of 3.4 on test graphs without affecting the quality of coloring graphs [20]. The CUDA implementation of the algorithm transfers a graph from a host to a GPU for parallel graph coloring and sends the colored graph back to the host for postprocessing. The CUDA kernels repeatedly process the vertices until convergence is reached. The CUDA implementation produces a deterministic coloring although the processing is done asynchronously for performance reason. Looking into the kernels, we find that the implementation is highly optimized with explicit warp-level programming [21]. We will focus on how the CUDA functions are mapped to SYCL in the next section.

2.2 The SYCL compiler with CUDA support

The initial approach to support NVIDIA computing platforms was based on the NVIDIA OpenCL 1.2 implementation [22]. The prototype demonstrated SYCL running on multiple platforms, but the capabilities of the OpenCL 1.2 implementation from NVIDIA are limited. Taking advantage of a plugin interface that can be selected at runtime by setting an environment variable [23], the new approach does not depend on the OpenCL support from NVIDIA,

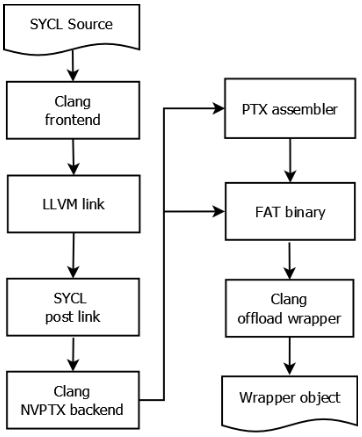


Figure 1: The Intel SYCL compiler with CUDA support [23]

facilitating more features and potentially higher overall performance. The plugin will support as many CUDA features as possible so that researchers and developers can migrate a wide variety of CUDA programs and execute them across vendors' GPU platforms successfully. In addition, SYCL developers have been addressing potential performance bottlenecks found in runtime, math functions, benchmarks, and scientific applications.

The SYCL compiler with CUDA support is built upon the SYCL branch of the Intel LLVM repository [24]. Figure 2 shows the device compile flow that allows for executing a SYCL program on a target GPU using the LLVM technologies [25, 26]. A SYCL compiler is not designed for converting a CUDA program to a SYCL program automatically. Instead, it enables researchers and developers to compile a SYCL program targeting an NVIDIA GPU.

3 EXPERIENCE OF MIGRATING THE CUDA IMPLEMENTATION

In this section, we will show the migration paths between CUDA and SYCL by explaining the CUDA and SYCL functions called in the implementations of the graph coloring algorithm.

3.1 Device property information

It may be desirable to query the device properties of a GPU to allocate its hardware resources efficiently at runtime. The CUDA device properties, which are defined in the "cudaDeviceProp" structure, can be queried using the "cudaGetDeviceProperties()" function in CUDA. In SYCL, a device can be queried for information by calling the "get_info()" member function of the SYCL "device" class, specifying an information parameter in "sycl::info::device". The CUDA program queries the multi-processor count (i.e., the number of streaming multiprocessors) and the number of maximum resident threads per multi-processor to determine the number of thread blocks for launching the CUDA kernels. The multi-processor count in CUDA can be mapped to the maximum number of compute-units in SYCL. Querying the clock rate of a CUDA device in KHz can be mapped to the maximum configured clock frequency of a device in MHz in SYCL. The number of maximum resident threads per multi-processor and the memory clock rate of a device do not have corresponding SYCL information parameters. A CUDA device's compute capability represented by a major revision number and a minor revision number can be queried with the version of the SYCL backend associated with the device. Table 1 lists the device information queried in the host program.

Table 1: Device information parameters in the CUDA and SYCL implementations

	CUDA	SYCL
1	`multiProcessorCount`	`sycl::info::device::max_compute_units`
2	`maxThreadsPerMultiProcessor`	N/A
3	`clockRate`	`sycl::info::device::max_clock_frequency`
4	`memoryClockRate`	N/A
5	`major/minor`	`sycl::info::device::backend_version`

3.2 Memory management using SYCL buffer

Two abstractions are commonly used for managing memory in SYCL: unified shared memory and buffer. The former is a pointer-based approach that allows for easier integration with existing C/C++ programs. In contrast, a buffer is considered as a high-level data abstraction because we can query characteristics of a buffer and determine whether and where device data is read from or written back to host memory. In this work, we focus on data management using SYCL buffer.

A SYCL buffer defines a shared array of one, two or three dimensions that can be accessed by a computationally intensive task (kernel). The content of a buffer is not directly accessed by a program. Accessing the underlying data in a buffer requires an accessor object. Such object also informs the runtime where and how data is accessed.

Table 2 lists a migration path from memory allocations in the CUDA program to buffer instances in SYCL. The CUDA function "cudaMalloc()" allocates "s × sizeof(T)" bytes of one-dimensional linear memory on the device and returns a pointer "p" to the allocated memory. There are "s" words of data type "T", and the size of each word is "sizeof(T)" bytes. In the SYCL program, buffers are instantiated with the specifications of data types, dimensions, and sizes of the underlying data in word. The initial content of the buffer is not specified. The constructed SYCL buffer will use a default allocator when allocating memory on the host. The size of the buffer is specified by the range parameter provided. Data is not written back to the host on destruction of the buffer unless the buffer has a valid pointer specified with the member function "set_final_data()". The memory management for this type of buffer is entirely handled by the SYCL runtime.

A SYCL buffer can also be constructed by passing a host pointer. The buffer is initialized with the memory pointed to by a host pointer "h". The ownership of this memory is given to the buffer for the duration of its lifetime. When the buffer is destroyed, the destructor will block until all work on the buffer have completed, then copy the content of the buffer back to the host memory if needed and then return.

In the author's opinion, constructing a SYCL buffer with a host pointer is elegant because it combines memory allocation and data copy from a host to a device. Additionally, taking a unit of word instead of a unit of byte for buffer construction may abstract away low-level details of memory allocation.

Memory spaces, which are allocated by previous calls to "cudaMalloc()", are released explicitly in the CUDA program. In contrast, the SYCL runtime will free any storage required for the buffers when they are no longer in use. This may improve programming productivity by relieving developers of manual memory deallocation in a complex program. However, understanding the implications of buffer destruction is important.

In the CUDA program, device memory is also allocated in global scope using the "__device__" declaration specifier. To map such memory to SYCL, a SYCL buffer is added as if the global memory were dynamically allocated with "cudaMalloc()". The size of the buffer is one for a variable or the length of an array.

Table 2: Memory management using the CUDA and SYCL functions.

	CUDA	SYCL
1	`cudaMalloc(&p, s*sizeof(T))`	`sycl::buffer<T, 1> d (s)`

2	`cudaMalloc(&p, s*sizeof(T))`
	`cudaMemcpy(p, h, s*sizeof(T),cudaMempyHostToDevice)`
		`sycl::buffer<T, 1> d (h, s)`

3	`cudaFree(p)`	`Released by the SYCL runtime`
4	`__device__ T var`	`sycl::buffer<T, 1> var (1)`

3.3 ND-Range

An N-dimensional range (ND-Range) is a natural way to invoke computation across elements in a domain such as a vector, matrix, or volume. All threads in a thread block or work-items in a work-group execute the same kernel program or instance in a single-program-multiple-data style. Each work-item may query its local and/or global location in groups that contain it and invoke group-specific functionalities. The execution of an ND-Range kernel in SYCL is consistent with the OpenCL execution model. The ND-Range covers the total execution range, which is divided into work-groups whose size must divide the ND-Range size in each dimension. Each work-group can be divided into sub-groups. Sub-group functionality is often leveraged to optimize the execution of work-items in a work-group at the level of hardware thread execution. Work-items in a sub-group can exchange data using hardware registers instead of local shared memory. On the other hand, sub-group functionality is not necessarily supported by all hardware devices from the same vendor or different compilers [6].

The SYCL specification has been improving functionality for groups of work-items, such as group barriers and collective operations. A collective function represents an operation performed by a group of work-items. These group functions act as synchronization points and must be reached by all work-items in the group before they move on. When one work-item in a group calls a group function, all work-items in that group must call the same function under the same conditions (e.g., in the same iterations of a loop). The group argument in the function indicates that all work-items in the specified group work together for a specific operation.

Table 3 lists a migration path from the warp-level primitives called in the CUDA program to the SYCL group functions. The warp vote functions in CUDA take as input an integer predicate from each thread in a warp and compare these values with zero. Results of the comparisons are reduced across the active threads of the warp in “any”, “all” or “ballot” logic. The result is then broadcasted to each participating thread. In contrast, the SYCL group functions require a sub-group argument “sg” that represents the sub-group to which each work-item belongs. For the “mask” argument in the CUDA warp vote functions, the SYCL “mask” is bitwise ANDed with a bit pattern from each work-item in a sub-group before it is logically ANDed with a Boolean predicate. When the value of a “mask” is 0xFFFFFFFF (i.e., 32 active threads), we may optimize away the bitwise operation and the final value is “pred”. The CUDA “__ballot_sync” primitive is mapped to the SYCL “reduce_over_group” function in which a group sums up values across a sub-group and each work-item provides one value.

The CUDA warp shuffle instruction “__shfl_sync” is mapped to the SYCL “select_from_group” function that allows work-items to obtain a copy of a value held by any other work-item in the group. The “__shfl_xor_sync” is mapped to the SYCL “permute_group_by_xor” function that permutes values by exchanging values held by pairs of work-items identified by computing the bitwise exclusive OR of the work-item identifier and a fixed lane mask.

The “mask” and “width” parameters in the CUDA warp shuffle instructions are not yet supported by the SYCL group functions. Compared to the CUDA ballot primitives, application and developers may prefer a concise way of calling a ballot primitive although a reduction operator over a group is not limited to the “+” operator in SYCL.

Table 3: The CUDA warp-level primitives and the SYCL group functions

	CUDA	SYCL
1	__any_sync(mask, pred)	sycl::any_of_group(sg, (mask & (1 << sg.get_local_linear_id())) && pred)
2	__all_sync(mask, pred)	sycl::all_of_group(sg, (mask & (1 << sg.get_local_linear_id())) && pred)
3	__ballot_sync(mask, pred)	sycl::reduce_over_group(sg, (mask & (1 << sg.get_local_linear_id())) && pred ? (1 << sg.get_local_linear_id())) : 0, plus<>())
4	__shfl_sync(mask, var, srcLane, width)	sycl::select_from_group(sg, var, srcLane)
5	__shfl_xor_sync(mask, var, laneMask, width)	sycl::permute_group_by_xor(sg, var, laneMask)

3.4 Arithmetic functions

Table 4 lists a migration path from the CUDA arithmetic functions invoked in the implementation of the algorithm to the SYCL arithmetic functions. The “max()” function in CUDA, which returns the maximum of two numbers, is mapped to the “sycl::max()” function. The “__clz()” intrinsic function in CUDA, which returns the number of consecutive high-order zero bits in a 32-bit integer, starting at the most significant bit (bit 31), is mapped to the “sycl::clz()” function. The “__ffs()” intrinsic function in CUDA finds the position of the least significant bit set to 1 in a 32-bit integer. When the integer’s value is zero, the function returns zero. The SYCL “sycl::ctz()” function counts the number of trailing zero bits in a number. When the value of the number is zero, the function returns the size in bits of the type of the number. Counting the trailing number of zero bits starting at the most significant bit is equivalent to finding the position of the least significant bit set to 1, but the discrepancy of the return values of the CUDA and SYCL functions when the number is zero should be considered. It should be pointed out that “__ctz()” is not defined in the CUDA programming guide whereas “sycl::ffs()” is not defined in the SYCL specification. The “__popc()” intrinsic function in CUDA, which counts the number of bits that are set to 1 in a 32-bit integer, is mapped to the “sycl::popcount()” function.

Table 4: The arithmetic functions in the CUDA and SYCL programs

	CUDA	SYCL
1	`max(x, y)`	`sycl::max(x, y)`
2	`__clz(x)`	`sycl::clz(x)`
3	`__ffs(x)`	`x == 0 ? 0 : sycl::ctz(x)`
4	`__popc(x)`	`sycl::popcount(x)`

3.5 Atomic functions

Atomic operations enable concurrent memory accesses from work-items in work-groups to a memory location without introducing data race. They guarantee that multiple updates to a memory location do not overlap, but the order of updates is not deterministic. They are commonly used in many parallel computing programs.

Table 5 lists a migration path from CUDA atomic functions invoked in the implementation of the graph coloring algorithm to the SYCL atomic class. The “sycl” namespace is omitted for clarity. The constructor for a SYCL atomic function accepts a variable of the SYCL multi_ptr<int> type. Because the atomic functions operate on integer values in global memory space in the CUDA program, the raw pointer “x” is cast to a global pointer, converted to a SYCL atomic object, and called with the operator-specific “fetch” method of the SYCL atomic class.

The “atomic_ref” class, which is defined in the SYCL 2020 specification, extends the atomic operations with memory orders and scopes. However, comparing the CUDA and SYCL atomic functions shows that the CUDA atomic functions are most concise while the SYCL atomic references are most verbose. Application developers may prefer a concise way of expressing atomic operations over a memory location in a SYCL program.

Table 5: The integer atomic functions in the CUDA and SYCL programs

	CUDA	SYCL
1	`atomicAdd(int* x, int var)`	`atomic<int>(global_ptr<int>(x)).fetch_add(var)`
2	`atomicAnd(int* x, int var)`	`atomic<int>(global_ptr<int>(x)).fetch_and(var)`

3.6 Kernel attribute

A kernel attribute is used to annotate a kernel to influence code generation by a SYCL device compiler. In the CUDA implementation, the number of work-items in a warp is 32 by default. To inform the SYCL compiler that the kernel must be compiled and executed with the specified sub-group size of 32 on an NVIDIA GPU, the SYCL-specific kernel attribute “[[sycl::reqd_sub_group_size(32)]]” is required. The attribute is shown in Table 6.

3.7 Kernel launch and definition

Table 6 lists a migration path from the execution of one of the CUDA kernels in the graph coloring application to that of a SYCL kernel. A CUDA kernel starts with the “__global__” declaration specifier. The number of thread blocks in a grid (“grid”) and the number of threads per block (“block”) which will execute a kernel are specified using a “<<<...>>>” execution configuration syntax. In SYCL, the body of a C++ lambda function represents a kernel and variables captured by value will be passed to the kernel as arguments. The “submit” method of a SYCL queue object is invoked to submit a data-parallel kernel to be executed on a device associated with the queue object. The number of thread blocks in a grid and the number of threads per block in CUDA are converted to the global work size (“gws”) and local work size (“lws”) using the SYCL “range” class, respectively. The number of threads per block equals the local work size, and the global work size is the product of the number of thread blocks and the number of threads per block. While SYCL uses work-items, local work size and global work size to describe its thread hierarchy, the number of work-groups in SYCL is equal to the number of thread blocks in CUDA. These work-groups can execute independently on a GPU. In the SYCL code, the “init” function is called inside a lambda function. Though this is not required, it attempts to minimize code changes when migrating a CUDA kernel.

Launching a SYCL kernel is verbose compared to the CUDA approach. This will increase lines of code and decrease programming productivity when there are many kernels in a large application. On the other hand, it offers the flexibility of combining host and device codes in a single source. There is a tradeoff between verbosity and flexibility in the SYCL programming model.

Table 6: Kernel launch in CUDA and SYCL

CUDA	SYCL
`__global__ void init (…) {`	`void init (…) {`
`  // kernel code`	`  // kernel code`
`}`	`}`
`init <<<grid, block>>> (…);`	`sycl::range<1> gws (grid*block);`
	`sycl::range<1> lws (block);`
	`q.submit([&](sycl::handler &cgh) {`
	`  // accessors are omitted`
	`  cgh.parallel_for(sycl::nd_range<1>(gws, lws)`
	`  [=] (sycl::nd_item<1> item)`
	`  [[sycl::reqd_sub_group_size(32)]] {`
	`    init(…) // call the “kernel” function`
	`  });`
	`});`

3.8 Debugging

The CUDA in-kernel “printf()” function, which is used for debugging kernel execution, behaves in a similar way to the standard C-library “printf()” function. Although the function is handy, it is not part of the SYCL specification. Instead, the SYCL “stream” class is a buffered output stream that allows displaying the values of built-in, vector and SYCL types to the console. It should be stressed that the stream class is designed for debugging purposes and should therefore be avoided for performance critical applications. In the SYCL implementation of the graph coloring algorithm, streaming output is disabled by default with the “#ifdef” directives.

3.9 Architecture-specific features

As far as we know, certain architecture-specific features in the CUDA program have no SYCL equivalents. To aid the compiler with additional information about register usage of the CUDA kernels, the CUDA program uses the “__launch_bounds__()” qualifier in the definition of a “__global__” function to specify the maximum number of threads per block with which to launch the kernel and the desired number of resident blocks per multiprocessor. The specification of thread and thread block counts at the SYCL kernel scope is not supported yet. The CUDA program sets the preferred cache configuration with “cudaFuncSetCacheConfig()” for devices that share the L1 cache and shared local memory. This is also not supported by the SYCL compiler.

4 EXPERIMENTAL RESULTS

4.1 Kernel execution time on NVIDIA GPUs

As mentioned in the last section, certain device-specific features are not yet supported by the SYCL compiler. When comparing the executing time of CUDA and SYCL kernels, the CUDA program contains these device-specific features. In the CUDA program, the number of blocks per grid is computed at runtime:

$$\text{Blocks} = \text{SMs} \times \text{maxThreadsPerMultiProcessor} \div \text{ThreadsPerBlock} \tag{1}$$

Since the number of streaming multiprocessors (SMs) is a SYCL device property and the number of threads per block (ThreadsPerBlock) is a constant value specified in the program, the value of “maxThreadsPerMultiProcessor” is specified explicitly in the SYCL program.

We evaluate the performance of the CUDA and SYCL kernels with 18 test graphs [27] on three GPU computing platforms [28]. The characteristics of the graph set are listed in Table 7. These graphs are selected for their variety in characteristics though coloring them does not necessarily make sense. The “oswald00” node contains an Intel Xeon E5-2683 v4 CPU and an NVIDIA Tesla P100 GPU. There are 56 SMs in the GPU. The “leconte” node contains IBM Power9 CPUs and NVIDIA Tesla V100-SXM2 GPUs. The “equinox” node contains an Intel Xeon E5-2698 v4 CPU and NVIDIA Tesla V100-DGXS GPUs. Both GPUs have 80 SMs. The thread block size is 512 and the value of “maxThreadsPerMultiProcessor” 2048 for the three GPUs. Hence, the number of blocks per grid are 224 and 320 for the P100 and V100, respectively. We present our results using a single GPU on each platform.

There are three kernels in the CUDA implementation of the graph coloring algorithm. We measure the average kernel execution time of 100 runs using the NVIDIA Nsight System [29]. We compile the CUDA program with the NVIDIA HPC SDK 22.1 and build the SYCL compiler with CUDA support from the source. The optimization option is “-O3” and the device-specific option is “-arch=sm_XY” where “XY” is 60 and 70 for the P100 and V100 GPUs, respectively. The GPU results are verified on the hosts.

Table 7: Names, types, vertex and edge counts, average and maximum degrees of a vertex in each graph

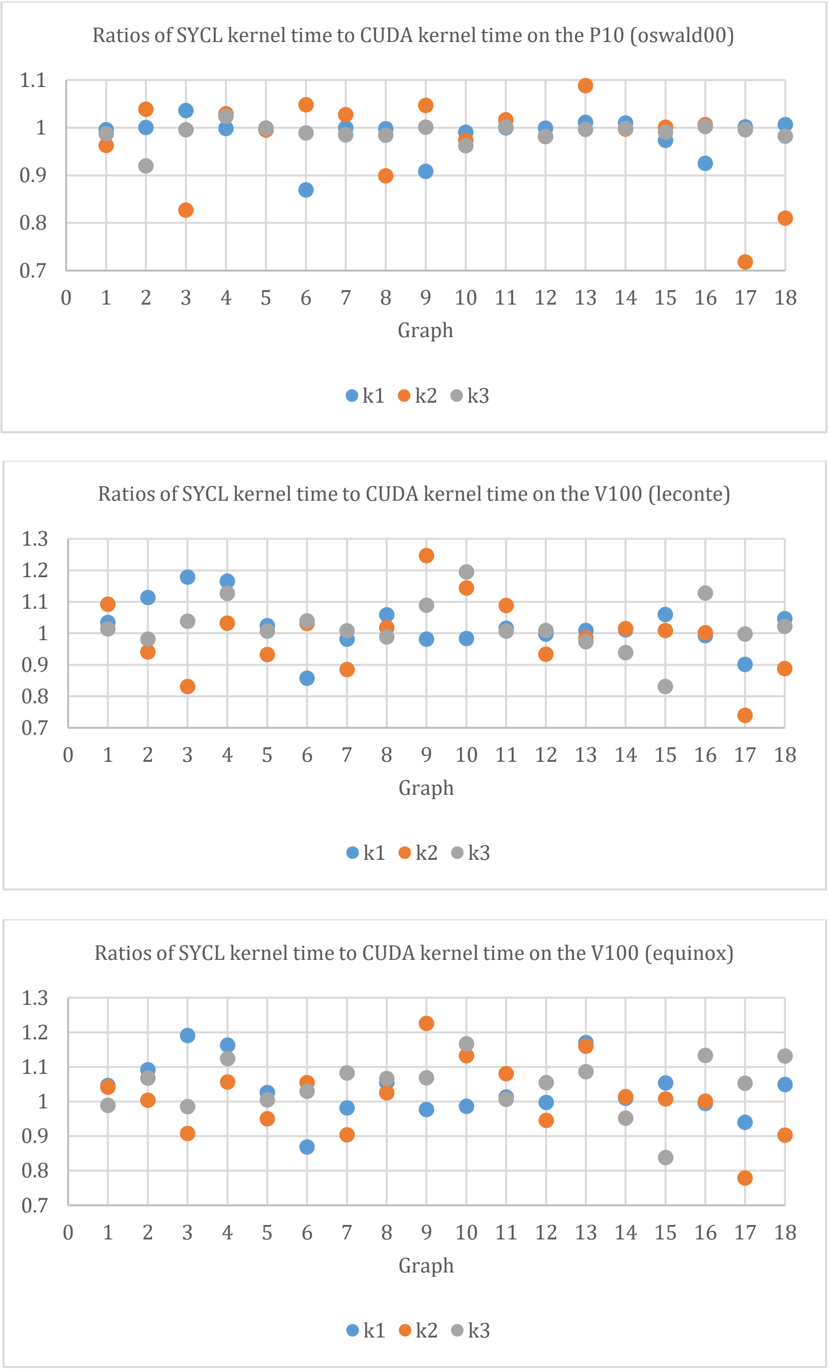
No.	Graph name	Type	Vertices	Edges	Degree$_{avg}$	Degree$_{max}$
1	2d-2d20.sym	Grid	1,048,576	4,190,208	4	4
2	amazon0601	Co-purchases	403,394	4,886,816	12.1	2752
3	as-skitter	Internet topo.	1,696,415	22,190,596	13.1	35455
4	citationCiteseer	Publication	268,495	2,313,294	8.6	1318
5	cit-Patents	Patent cites	3,774,768	33,037,894	8.8	793
6	coPapersDBLP	Publication	540,486	30,491,458	56.4	3299
7	delaunay_n24	Triangulation	16,777,216	100,663,202	6	26
8	europe_osm	Road map	50,912,018	108,109,320	2.1	13
9	in-2004	Web links	1,382,908	27,182,946	19.7	21869
10	internet	Internet topo.	124,651	387,240	3.1	151
11	kron_g500-logn21	Kronecker	2,097,152	182,081,864	86.8	213904
12	r4-2e23.sym	Random	8,388,608	67,108,846	8	26
13	rmat16.sym	RMAT	65,536	967,866	14.8	569
14	rmat22.sym	RMAT	4,194,304	65,660,814	15.7	3687
15	soc-LiveJournal1	Community	4,847,571	85,702,474	17.7	20333
16	uk-2002	Web links	18,520,486	523,574,516	28.3	194955
17	USA-road-d.NY	Road map	264,346	730,100	2.8	8
18	USA-road-d.USA	Road map	23,947,347	57,708,624	2.4	9

Figures 2 show the ratios of the execution time of the SYCL kernels to that of the CUDA kernels for the graph set on these systems. When the ratio is over 1, the SYCL kernel takes longer time. For each test graph numbered from 1 to 18, the kernel "k1" takes the longest time while the kernel "k3" takes the shortest time. "k1" does not necessarily correspond to the same kernel in the source file. On "oswald00", the ratio ranges from 0.718 to 1.088. On "leconte", the ratio ranges from 0.739 to 1.246. On "equinox", the ratio ranges from 0.778 to 1.225. The overall performance trends with respect to the graph set on the GPUs show that the SYCL kernels could achieve comparable performance using the SYCL compiler with CUDA support.

4.2 CUDA API statistics

In Table 8, the CUDA API statistics from the profiler show that the SYCL program invokes the CUDA driver APIs for context construction, memory allocation, data transfer, synchronization, etc. These APIs are called in the CUDA plugin interface in the SYCL compiler for fine-grain control over the implementation of the SYCL runtime. In contrast, only CUDA runtime APIs are displayed when the CUDA program is profiled. Most application developers would write CUDA programs with the runtime APIs. They provide implicit initialization, context management, and module management for simpler code.

The names and number of calls of the CUDA APIs invoked by the CUDA and SYCL programs are listed in Table 8. The API statistics for the SYCL program show that the SYCL runtime will release eight SYCL buffers, and that there are two memory copies from host to device and one memory copy from device to host. The events are created to synchronize data transfers with kernel execution.


Ratios of SYCL kernel time to CUDA kernel time on the P10 (oswald00)

Ratios of SYCL kernel time to CUDA kernel time on the V100 (leconte)

Ratios of SYCL kernel time to CUDA kernel time on the V100 (equinox)

Figure 2: Ratios of kernel execution time on the three NVIDIA GPUs

Table 8: CUDA API statistics for the CUDA and SYCL programs

CUDA	SYCL
`cudaMalloc (8)`	`cuCtxCreate_v2 (1)`
`cudaDeviceSynchronize (2)`	`cuCtxDestroy_v2 (1)`
`cudaLaunchKernel (300)`	`cuLaunchKernel (300)`
`cudaMemcpy (3)`	`cuMemAlloc_v2 (8)`
`cudaFree (8)`	`cuMemFree_v2 (8)`
`cudaMemset (100)`	`cuMemcpyDtoHAsync_v2 (1)`
	`cuMemsetD32Async (100)`
	`cuMemcpyHtoDAsync_v2 (2)`
	`cuEventRecord (404)`
	`cuModuleLoadDataEx (1)`
	`cuEventCreate (404)`
	`cuEventDestroy_v2 (404)`
	`cuModuleUnload (1)`
	`cuEventSynchronize (32)`
	`cuStreamSynchronize (3)`
	`cuStreamCreate (1)`
	`cuStreamDestroy_v2 (1)`
	`cuCtxSynchronize (1)`

4.3 Performance evaluation on an Intel integrated GPU

Migrating the CUDA program to SYCL allows us to evaluate the program on an Intel GPU computing platform. In our experiment, we choose an Intel UHD Graphics P630 integrated GPU (iGPU) for executing the kernels with the

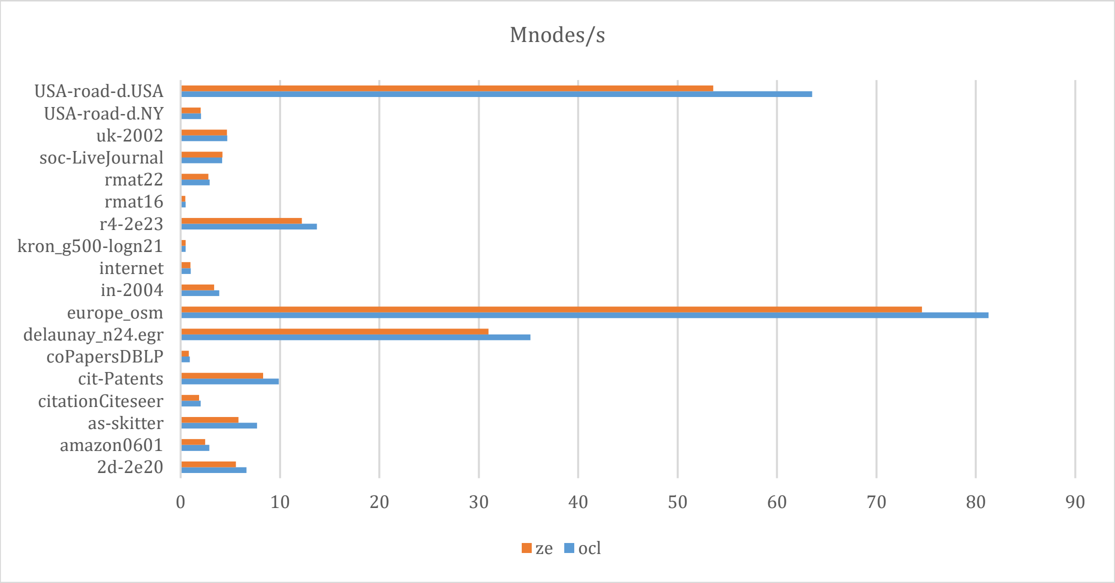


Figure 3: Throughput in millions of nodes per second with the OpenCL and Level Zero interfaces on the P630 iGPU

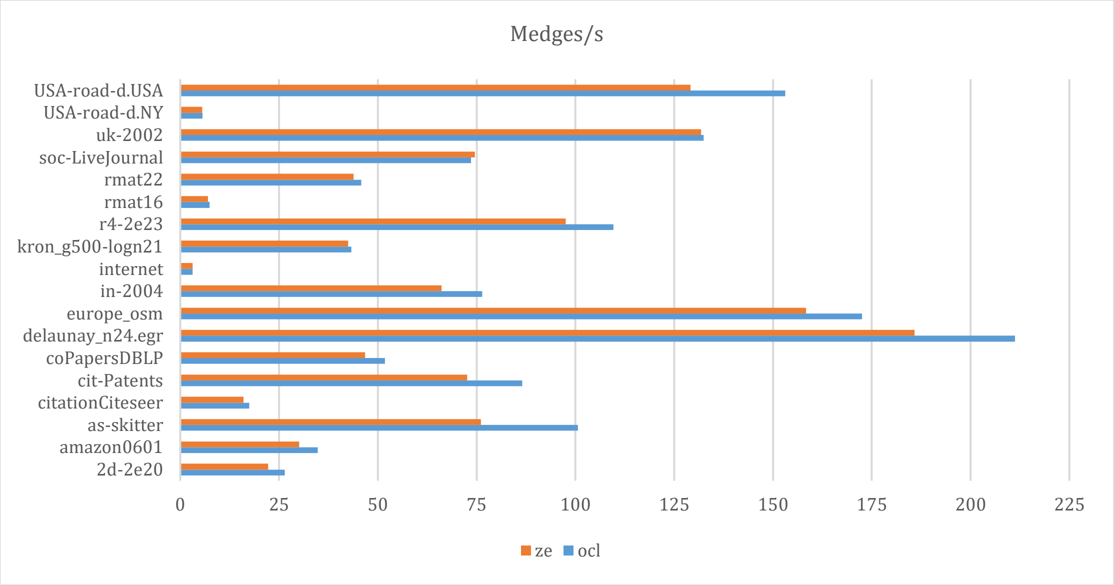


Figure 4: Throughput in millions of edges per second with the OpenCL and Level Zero interfaces on the P630 iGPU

OpenCL and Level Zero compute APIs [30]. The CPU is an Intel Xeon E-2176G, and the thermal design power (TDP) is 80 Watts [31]. Such class of GPUs, with a CPU and a GPU integrated on the same chip, is commonly used in laptops, desktop computers, and low-cost servers. Integrated GPUs are not designed to outperform discrete GPUs due to the power, area, and thermal constrains.

In the SYCL program, the number of threads per block (local work size) is 256, which is the maximum value supported by the target device. Because of the architectural differences between an Intel GPU and an NVIDIA GPU, the number of work-groups is specified directly, and its value is 21 for efficient resource utilization. We build the SYCL program with the Intel oneAPI Data Parallel C++ compiler, version 2022.0.0. The compiler optimization option is “-O3”. The performance metrics are the throughputs of completed (colored) nodes and edges.

Figure 3 shows the throughput in millions of completed nodes per second (Mnodes/s) and Figure 4 shows the throughput in millions of completed edges per second (Medges/s). The throughputs using the OpenCL interface (ocl) are on average 10.5% higher than those using the Level Zero interface (ze) for the graph set on the target device. An analysis of the host and device execution time indicates that the performance drop is caused by the “init” kernel. For example, the kernel is almost 2.5X slower using the Level Zero interface for the graph “USA-road-d.USA”.

Given the power constraint of the iGPU, we try to measure the average package power consumption of executing the SYCL program with the Intel SoC Watch application, version 2021.3. The minimum time interval between data collection points is 1 millisecond. Because of the overhead of polling sensor data during program execution on a device, the measured average package power will be higher than that of running a program with data collection disabled. Hence, the measured power is an estimate of the worst-case scenario.

We run the SYCL program four times and report the highest power in milliwatts (mW). Figure 5 shows that the average package powers across the graph set fall into a range from 37 W to 41 W. Hence, the measured average package power consumed by the application is approximately half of TDP.

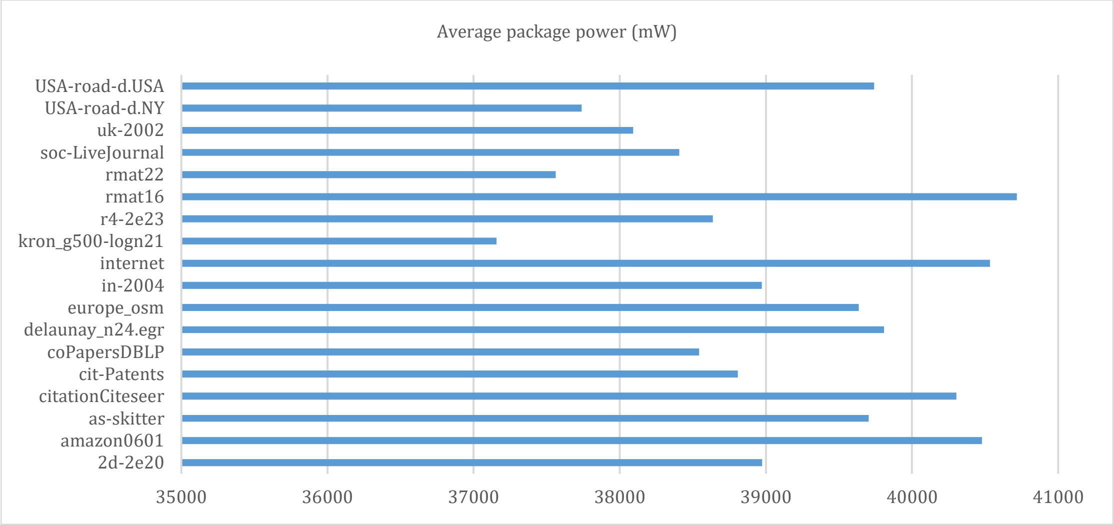


Figure 5: Average package power measured by the Intel SoC Watch on the Intel E-2176G CPU with a P630 iGPU

5 RELATED WORK

CUDA is a mature programming model for NVIDIA computing devices whereas SYCL has become a promising programming model for CPUs, GPUs, and other accelerators. In [32], the authors evaluate the performance of benchmarks and mini-apps having both SYCL and CUDA implementations on an NVIDIA Volta GPU. While there is missing functionality support, the performance of running SYCL is competitive with using CUDA directly. Many of the performance differences are due to the ordering and choices of how to load memory. In [8], the authors evaluate the performance of a GPU accelerated sequence alignment algorithm across multiple vendor GPUs and programming models. They describe the code changes required for the SYCL implementation to execute the application successfully. They conclude that porting their highly optimized CUDA kernels to SYCL requires significant code changes. The performance of the SYCL implementation is 2X slower than that of the CUDA implementation on the target devices. They mention the usage of the CUDA-to-SYCL conversion tool for automatic code translation [33], but the generated codes are unnecessarily complex and still require major changes. In [34], the authors evaluate the HPC applications written in OpenCL and SYCL on AMD, Intel, and NVIDIA GPUs and show that across each application the SYCL implementation achieves similar performance to a direct OpenCL implementation. In [35], the authors share their experience in creating mini-apps for the Wilson-Dslash stencil operator for Lattice Quantum Chromodynamics using the SYCL programming model. In their opinions, the SYCL way of managing memory through buffers and accessors are somewhat cumbersome and may create difficulties interfacing with non-SYCL external libraries in an efficient way. Sometimes, it is desirable to have explicit control over where the data is rather than delegating the management of memory to the SYCL runtime. In [36], the authors describe their customized porting flow for their platform-portable math library. They present a hierarchical view of CUDA and SYCL kernel calls and parameters for a clear understanding of the differences of the two programming models. The SYCL compiler did not support subgroup vote functions, so they emulated these functions and suggested native support of subgroup vote function for performance portability. With the active development of the SYCL compiler, we are now able to utilize the SYCL group functions for migrating CUDA warp-level primitives. In [37], the author shares his extensive experience of using SYCL for CUDA. While both programming models are extensions to the C/C++ languages, there are significant differences along a migration path between CUDA and SYCL. The optimizations applied by a compiler to a kernel also pose challenges and complexities to performance portability.

6 CONCLUSION

Integrating the underlying concepts, portability, and efficiency of OpenCL with the flexibility of single-source C++, SYCL is a promising programming model for heterogeneous computing devices. The CUDA plugin interface allows researchers and developers to execute a SYCL program on an NVIDIA GPU. Although the SYCL compiler offers CUDA support, a good understanding of the two programming models is still needed. In this work, we describe our experience of migrating an optimized parallel implementation of the graph coloring algorithm from CUDA to SYCL. While certain CUDA features are not supported by the SYCL specification, the experimental results show that the CUDA and SYCL kernels are comparable in terms of kernel performance over the graph set on the NVIDIA GPUs. Additionally, migrating the CUDA program to SYCL allows us to evaluate the application on an Intel GPU using multiple backend interfaces and an energy profiling tool. Porting CUDA applications to SYCL and evaluating functional and performance portability will be helpful for the growth of the SYCL ecosystem. With the active development of the SYCL compilers and applications, functional and performance portability will continue to be improved.

ACKNOWLEDGMENT

I appreciate the anonymous reviewers for their constructive criticisms. In addition, I want to thank Miroslav Stoyanov for his comments and suggestions. This research used resources of the Experimental Computing Lab at Oak Ridge National Laboratory and the Intel DevCloud. This research was supported by the US Department of Energy Advanced Scientific Computing Research program under Contract No. DE-AC05-00OR22725.

REFERENCES

[1] Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y. and Volkov, V., 2008. Parallel computing experiences with CUDA. IEEE MICRO, 28(4), pp.13-27.

[2] Munshi, A., Gaster, B., Mattson, T.G. and Ginsburg, D., 2011. OpenCL programming guide. Pearson Education.

[3] Kaeli, D., Mistry, P., Schaa, D. and Zhang, D.P., 2015. Heterogeneous computing with OpenCL 2.0. Morgan Kaufmann.

[4] Li, P., Brunet, E., Trahay, F., Parrot, C., Thomas, G. and Namyst, R., 2015, September. Automatic OpenCL code generation for multi-device heterogeneous architectures. In 2015 44th International Conference on Parallel Processing (pp. 959-968). IEEE.

[5] Steuwer, M. and Gorlatch, S., 2014. SkelCL: a high-level extension of OpenCL for multi-GPU systems. The Journal of Supercomputing, 69(1), pp.25-33.

[6] 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.

[7] Stroustrup, B., 2013. The C++ Programming Language. Pearson Education.

[8] Haseeb, M., Ding, N., Deslippe, J. and Awan, M., 2021, November. Evaluating Performance and Portability of a core bioinformatics kernel on multiple vendor GPUs. In 2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC) (pp. 68-78). IEEE.

[9] Leighton, F.T., 1979. A graph coloring algorithm for large scheduling problems. Journal of research of the national bureau of standards, 84(6), pp.489-506.

[10] Chaitin, G.J., 1982. Register allocation and spilling via graph coloring. ACM Sigplan Notices, 17(6), pp.98-101.

[11] Matula, D.W. and Beck, L.L., 1983. Smallest-last ordering and clustering and graph coloring algorithms. Journal of the ACM (JACM), 30(3), pp.417-427.

[12] Coleman, T.F. and Moré, J.J., 1983. Estimation of sparse Jacobian matrices and graph coloring blems. SIAM journal on Numerical Analysis, 20(1), pp.187-209.

[13] Hansen, P. and Delattre, M., 1978. Complete-link cluster analysis by graph coloring. Journal of the American Statistical Association, 73(362), pp.397-403.

[14] Garey, Michael R., and David S. Johnson. "Computers and Intractability", vol. 29. W. H. Freeman and Company, New York (2002), pp 1-99.

[15] Jones, M.T. and Plassmann, P.E., 1993. A parallel graph coloring heuristic. SIAM Journal on Scientific Computing, 14(3), pp.654-669.

[16] Çatalyürek, Ü.V., Feo, J., Gebremedhin, A.H., Halappanavar, M. and Pothen, A., 2012. Graph coloring algorithms for multi-core and massively multithreaded architectures. Parallel Computing, 38(10-11), pp.576-594.

[17] Cohen, J. and Castonguay, P., 2012, May. Efficient graph matching and coloring on the gpu. In GPU Technology Conference (pp. 1-10).

[18] Hasenplaugh, W., Kaler, T., Schardl, T.B. and Leiserson, C.E., 2014, June. Ordering heuristics for parallel graph coloring. In Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures (pp. 166-177).

[19] Singhal, N., Peri, S. and Kalyanasundaram, S., 2017, January. Practical multi-threaded graph coloring algorithms for shared memory architecture. In Proceedings of the 18th International Conference on Distributed Computing and Networking (pp. 1-7).

[20] Alabandi, G., Powers, E. and Burtscher, M., 2020, February. Increasing the parallelism of graph coloring via shortcutting. In Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (pp. 262-275).

[21] Hong, S., Kim, S.K., Oguntebi, T. and Olukotun, K., 2011. Accelerating CUDA graph algorithms at maximum warp. ACM Sigplan Notices, 46(8), pp.267-276.

[22] Reyes, R., Brown, G. and Burns, R., 2020, April. Bringing performant support for NVIDIA hardware to SYCL. In Proceedings of the International Workshop on OpenCL (pp. 1-1).

[23] https://github.com/intel/llvm/blob/sycl/sycl/doc/PluginInterface.md

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

[25] https://github.com/intel/llvm/blob/sycl/sycl/doc/CompilerAndRuntimeDesign.md

[26] Lattner, C. and Adve, V., 2004, March. LLVM: A compilation framework for lifelong program analysis & transformation. In International Symposium on Code Generation and Optimization, 2004. CGO 2004. (pp. 75-86). IEEE.

[27] https://userweb.cs.txstate.edu/~burtscher/research/ECLgraph/index.html

[28] https://excl.ornl.gov/

[29] Knobloch, M. and Mohr, B., 2020. Tools for GPU computing–debugging and performance analysis of heterogenous hpc applications. Supercomputing Frontiers and Innovations, 7(1), pp.91-111.

[30] https://github.com/intel/compute-runtime

[31] https://www.intel.com/content/www/us/en/products/sku/134860/intel-xeon-e2176g-processor-12m-cache-up-to-4-70-ghz/specifications.html

[32] Homerding, B. and Tramm, J., 2020, April. Evaluating the Performance of the hipSYCL Toolchain for HPC Kernels on NVIDIA V100 GPUs. In Proceedings of the International Workshop on OpenCL (pp. 1-7).

[33] Jin, Z. and Vetter, J., 2021, June. Evaluating CUDA Portability with HIPCL and DPCT. In 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (pp. 371-376). IEEE.

[34] 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).

[35] 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.

[36] Tsai, Y.M., Cojean, T. and Anzt, H., 2021. Porting a sparse linear algebra math library to Intel GPUs. arXiv preprint arXiv:2103.10116.

[37] Migdal, M. 2021. From CUDA to SYCL. SYCL summer sessions. https://sycl.tech/assets/files/Michel_Migdal_Codeplay_Porting_Tips_CDUA_To_SYCL.pdf.