Experience Deploying Graph Applications on GPUs with SYCL

Abstract. SYCL allows for deployment and use of accelerators across vendors' platforms. In this work, we describe the experience of deploying graph analytics on vendors' GPUs using SYCL. We contrast the CUDA and SYCL application programming interfaces by describing the experience of migrating the applications from CUDA to SYCL, evaluate the performance of the applications on NVIDIA and AMD GPUs, and explore performance improvement with device-level parallelism. The results show that the recent SYCL extensions facilitate functional portability, but improving code optimizations and resource usage for performance portability is needed in the compiler implementation.

1 INTRODUCTION

CUDA has enabled wide use of graphics processing units (GPUs) as an accelerator for computationally intensive tasks [1]. However, it is a proprietary programming model mainly optimized for NVIDIA GPUs. In contrast, Open Computing Language (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 tasks to heterogeneous computing devices [2, 3]. An OpenCL program can execute on a variety of platforms, but porting a program from CUDA to OpenCL tends to be error-prone and time-consuming [4, 5]. Built on the underlying concepts, portability, and efficiency of OpenCL, SYCL is a specification that defines a single-source abstract layer in C++ on top of OpenCL [6, 7]. The abstraction in SYCL could facilitate migrating CUDA programs although a typical SYCL platform still consists of a host connected to one or more vendors' devices as shown in Figure 1.

It is desirable to deploy a SYCL application across vendors' computing platforms without much loss of performance. However, achieving performance portability is challenging due to the characteristics of applications, target platforms, and maturity of a compiler. Despite these factors, there is always a need to identify applications where performance can be improved with the development of SYCL. Toward the goal of contributing to the SYCL ecosystem in the deployment of SYCL applications across vendors' platforms, we describe the experience of deploying graph analytics on GPUs using SYCL. More specifically, we choose optimized CUDA implementations of the graph algorithms, migrate the implementations from CUDA to SYCL manually, compile them with an open-source SYCL compiler, and evaluate the performance of the compute kernels in the applications on NVIDIA and AMD GPUs.

Here is a summary of our findings. More SYCL extensions have been added to the Intel SYCL compiler, an open-source implementation of the SYCL specification [8]. These extensions are not necessarily the core SYCL features

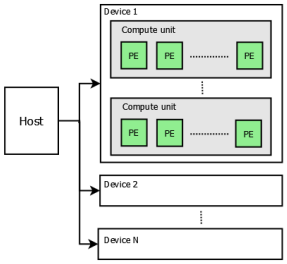

Figure 1. A SYCL platform with multiple devices. Each device contains multiple compute units. A compute unit is composed of one or more processing elements (PEs).

defined in the specification, but they facilitate migrating applications from CUDA to SYCL. Using the ratio of the raw performance of the SYCL applications to the performance of the applications written in a native programming model as a performance metric, we observe that the ratio ranges from 0.58 to 1 on an AMD RX6900 XT GPU and from 0.989 to 1.024 on an NVIDA GeForce RTX3090 GPU for graph coloring. For connected components, the ratios range from 0.659 to 0.986 and from 0.868 to 1.16 on the AMD and NVIDA GPUs, respectively. Furthermore, we explore device-level parallelism by executing kernels in each application on the AMD and NVIDA GPUs simultaneously. The performance speedup ranges from 0.56 to 1.99 for graph coloring and from 0.6 to 1.897 for connected components. The speedups we could obtain depend on the raw performance of the application on each GPU. We have described the motivation and scope of our study in this section. Section 2 briefly introduces the graph applications and the compiler in our experiment. Section 3 contrasts the CUDA and SYCL applications by describing in detail the migration paths from CUDA to SYCL. Section 4 presents the experimental results on the GPUs. Section 5 is a summary of related work, and Section 6 concludes the paper.

2 BACKGROUND

2.1 Brief introduction to the graph applications

Graph analytics algorithms such as graph coloring and connected components are widely used in many domains [9, 10, 11, 12, 13, 14, 15, 16]. Graph coloring 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. The problem is NP-hard, so there is no known polynomial time algorithm that can solve it optimally [17]. Heuristic algorithms can color a graph with no adjacent vertices assigned the same color, but they may require more colors than the optimal algorithm [18, 19, 20, 21, 22]. Connected components computes maximal subgraphs of an undirected graph such that there exists a path between any pair of vertices in the subgraph but there is no path between any pair of vertices from different subgraphs [23, 24]. In this work, we choose the highly optimized implementations of the algorithms that exploit the thread-, warp-, and block-level parallelism exposed in CUDA [25, 26, 27]. Hence, they are considered as hybrid implementations optimized to reduce load imbalance and to exploit hardware parallelism. The applications transfer a graph from a host to a GPU for parallel processing and send the result back to the host for postprocessing and validation.

2.2 Brief introduction to the SYCL compiler with CUDA and HIP support in our experiment

In this work, we choose an open-source SYCL compiler from the SYCL branch of the Intel LLVM GitHub repository [28, 29] for evaluating the applications. The initial approach to supporting NVIDIA computing platforms in the SYCL compiler was based on the NVIDIA OpenCL 1.2 implementation [30]. The prototype demonstrated the success of running on multiple platforms, but the capabilities of the OpenCL implementation from NVIDIA are limited. Taking advantage of a plugin interface that can be selected at runtime [31], the new approach does not depend on the OpenCL support from NVIDIA, facilitating extensions to more features and potentially higher overall performance. To support AMD GPUs, the CUDA plugin is migrated to the heterogeneous interface for portability (HIP) plugin with the support of Radeon Open Computing Platform [32]. The CUDA and HIP plugins have seen improvement in functionality and performance with the evaluation of SYCL applications and benchmarks from the community.

3 CONTRAST THE CUDA AND SYCL APPLICATIONS

A contrast of the two programming models from the aspect of application programming interfaces allows for a good understanding of their differences in device query, memory management, arithmetic and atomic functions, and kernel execution.

3.1 Device property query

It may be desirable to query the device properties of a GPU for allocating its hardware resources 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 a parameter related to the query. The CUDA applications query 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 is mapped to the maximum number of compute-units in SYCL. Querying the clock rate of a GPU device in KHz can be mapped to the maximum configured clock frequency of a device in MHz in SYCL. The GPU memory clock rate is mapped to a vendor-specific extension to device information. Due to the lack of compiler support, we map the number of maximum resident threads per multi-processor in CUDA to the maximum work-item size per compute unit by implementing the query in the SYCL compiler. 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 applications. For clarity, we omit the full namespace for each device parameter in SYCL.

Table 1: Contrast the CUDA device properties with SYCL device information queries in the applications

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

3.2 Memory management

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

Table 2: Contrast the CUDA and SYCL memory management and data transfers in the applications

CUDA	SYCL
`cudaMalloc(&dst, numBytes);`	`T* dst = sycl::malloc_device<T>(count, q);`
`cudaMemcpy(dst, src, numBytes), cudaMempyHostToDevice);`	`q.memcpy(dst, src, numBytes);`
`cudaMemcpy(dst, src, numBytes), cudaMempyDeviceToHost);`	`q.memcpy(dst, src, numBytes);`
`cudaFree(p);`	`sycl::free(p, q);`
`__device__ T var;`	`T *var = sycl::malloc_device<T>(1, q);`

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. Since the pointer-based approach is much closer to how memory is handled by CUDA, we will choose unified shared memory for managing memory resources and data transfers between a host and a device.

Table 2 lists the programming interfaces for memory management and data copy in CUDA and SYCL. In CUDA, "cudaMalloc()" allocates one-dimensional linear memory on a device in bytes and returns a pointer to the allocated memory. In SYCL, a templated function is called with the word size and a SYCL queue object "q" as the parameters. Hence, a double pointer is not needed for allocating device memory from a programmer perspective. Compared to the CUDA memory copy function that explicitly specifies the kind of transfer, the copy direction is implied by the types of source and destination memories in SYCL. Releasing device memory in SYCL is similar to memory deallocation in a C program, but the function requires a SYCL queue object associated with the allocated memory. In the CUDA applications, device memory is also statically allocated in global scope using the "__device__" declaration specifier. Neither the SYCL specification nor the SYCL compiler supports such specifier. Hence, we explicitly allocate device memory of length 1.

3.3 Group functions

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 contrasts the CUDA warp-level primitives and the SYCL group functions called in the applications. 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 computed from each work-item in a sub-group before it is logically ANDed with a Boolean predicate. When the value of a "mask" is

Table 3: Contrast the CUDA warp-level primitives with the SYCL group functions in the applications

CUDA	SYCL
`__any_sync(mask, pred)`	`sycl::any_of_group(sg, (mask & (1 << sg.get_local_linear_id())) && pred)`
`__all_sync(mask, pred)`	`sycl::all_of_group(sg, (mask & (1 << sg.get_local_linear_id())) && pred)`
`__ballot_sync(mask, pred)`	`auto mask = sycl::group_ballot(sg, pred); mask.extract_bits(mask_bits, 0)`
`__shfl_sync(MASK, var, srcLane)`	`sycl::select_from_group(sg, var, srcLane)`
`__shfl_xor_sync(MASK, var, laneMask)`	`sycl::permute_group_by_xor(sg, var, laneMask)`

0xFFFFFFFF (i.e., 32 active threads), we may optimize away the bitwise operation. Previously, the CUDA "__ballot_sync" primitive was 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 new SYCL "group_ballot" function converts a Boolean condition from each work-item in the group into a group mask (object). When a work-item's predicate is true, a bit corresponding to the work-item is set in this mask. The "extract_bits" method of the object is needed to return the values of these bits from the mask.

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 value of the mask (MASK) is 0xFFFFFFFF in the applications.

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()" or "min()" function in CUDA, which returns the maximum or minimum of two numbers, is mapped to the "sycl::max()" or "sycl::min()" 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

Table 4: Contrast the CUDA and SYCL arithmetic functions in the applications

CUDA	SYCL
`max(x,y) or min(x,y)`	`sycl::max(x,y) or sycl::min(x,y)`
`__clz(x)`	`sycl::clz(x)`
`__ffs(x)`	`x == 0 ? 0 : sycl::ctz(x)`
`__popc(x)`	`sycl::popcount(x)`

Table 5: Contrast the CUDA atomic functions with the SYCL atomic references in the applications

CUDA	SYCL

`atomicAdd(int* x, int var)`	`auto a = atomic_ref<int,`
	`memory_order::relaxed,`
	`memory_scope::device,`
	`address_space::global_space>(*x);`
	`a.fetch_add(var)`

`atomicCAS(int *x,`
`int expected,`
`int desired)`
	`int expected_value = expected;`
	`auto a = atomic_ref<int,`
	`memory_order::relaxed,`
	`memory_scope::device,`
	`address_space::global_space>(*x);`
	`a.compare_exchange_strong(expected_value, desired);`
	`return expected_value;`

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.

3.5 Atomic functions

Atomic operations enable concurrent memory accesses from multiple work-items in work-groups to a memory location without introducing data race in the applications. They guarantee that multiple updates to a memory location do not overlap, but the order of updates is not deterministic. We find that the application programming interfaces for atomic functions differ significantly between CUDA and SYCL.

Table 5 lists the CUDA and SYCL atomic add and atomic compare and swap (exchange) functions invoked in the implementations of the graph analytics algorithms. For the applications, the atomic operations are performed over 32-bit integer values stored in global device memory. The CUDA atomic add function reads the 32-bit word "old" located at the address "x" in global memory, compute the sum, and stores the result to memory at the same address. These three operations are performed in one atomic transaction. The function returns "old". The SYCL "atomic_ref" class, defined in the SYCL 2020 specification, extends the atomic operations with memory orders and scopes. The "add" function atomically sums an operand and the value of the object referenced and assigns the result to the value of the referenced object. The CUDA atomic compare and swap function reads the 32-bit word "old" located at the address "x" in global memory, computes "(old == expected ? desired : old)", and stores the result back to memory at the same address. The function returns the value "old". The SYCL compare and exchange function atomically compares the value of the object referenced against the value of expected. If the values are equal, the value of the referenced object is replaced with the value of desired; otherwise assigns the original value of the referenced object to expected. The function returns a Boolean value of "true" if the comparison operation was successful.

3.6 Kernel attribute

A kernel attribute annotates 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, 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 the execution of one of the GPU kernels in CUDA and SYCL. Other kernels can be launched in a similar fashion. 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

Table 6: Contrast the CUDA and SYCL kernel execution

CUDA	SYCL

`__global__ void init (…) {`
`  // kernel code`
`}`
	`void init (…) {`
	`  // kernel code`
	`}`

`dim3 grid (numBlocks)`
`dim3 block(threadsPerBlock)`
`init <<<grid, block>>> (…);`
	`sycl::range<1> gws (numBlocks * threadsPerBlock);`
	`sycl::range<1> lws (threadsPerBlock);`
	`q.submit([&](sycl::handler &cgh) {`
	`  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`
	`  });`
	`});`

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 device. In the SYCL code, the "init" function is called inside a lambda function. Though this is not required, it could minimize code changes when mapping a kernel from CUDA to SYCL.

Launching a SYCL kernel is verbose compared to the CUDA approach. This increases lines of code and decreases 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.

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 for displaying the values of built-in, vector and SYCL types to the console. The SYCL stream is designed for debugging purposes only and should therefore be avoided for performance critical applications. On the other hand, we find that the C function is only supported by the CUDA backend of the SYCL compiler.

3.9 Architecture-specific features

As far as we know, certain architecture-specific features in the CUDA applications have no SYCL equivalents though extensions are being added to the compiler implementation. 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 the thread block counts at the SYCL kernel scope is not supported by the compiler yet.

The CUDA applications set the preferred cache configuration with "cudaFuncSetCacheConfig()" for GPU devices that share the level-1 cache and shared local memory (SLM). To facilitate the migration process, SYCL recently

introduces a cache configuration property for specifying the division between the cache and local memory [33]. The value of the property is either "large_slm" or "large_data". The former prefers larger shared local memory to smaller L1 data cache. The latter prefers larger L1 data cache and smaller shared local memory. The new feature is an experimental extension specification, intended to provide early access to features and gather community feedback. The property may be ignored by GPU backends that do not support this extension.

4 EXPERIMENTAL RESULTS

4.1 Performance evaluation on the GPUs

We evaluate the performance of the applications with an open-source graph set [34]. 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. We offload the compute kernels in the applications to a compute node equipped with an NVIDIA GeForce RTX3090 GPU and an AMD RX6900 XT GPU. The CUDA and HIP programs are compiled with the NVIDIA HPC SDK 22.11 and ROCm 5.4 [30], respectively. We build the SYCL compiler with CUDA and HIP support from the source (2023-05-01). The optimization option is "-O3". All GPU results are verified on the hosts.

In the CUDA programs, the number of thread blocks per grid is determined at runtime as follows:

$$\text{Blocks} = \text{SMs} \times \text{maxThreadsPerMultiProcessor} \div \text{ThreadsPerBlock} \quad (1)$$

In the expression, the number of streaming multiprocessors (SMs) can be queried at runtime and the number of threads per block (ThreadsPerBlock) is a constant value specified in the program. The thread block (work-group)

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

size is fixed at 256. The maximum numbers of resident threads per multi-processor are 1536 and 2048 for the RTX3090 and RX6900, respectively. The performance metrics are million nodes processed per second (Mnodes/s) and million edges processed per second (Medges/s). We choose the maximum performance results among four trial runs. Each run averages the performance of executing the compute kernels for 100 times.

Tables 8 and 9 list the performance of the two applications running on the GPUs, respectively. We observe that the performance variances depend on the characteristics of the input graphs, the programming models selected for implementing the algorithms, and the GPU devices. For each input graph, we compare the performance of the SYCL implementation with that of the implementation using a native programming model (CUDA and HIP) on each GPU for evaluating performance portability.

To visualize the results, Figure 2 shows the ratios of the raw performance of the SYCL applications to that of the applications using native languages. When the ratio is above 1, the performance is higher for the SYCL implementation. For graph coloring, the ratios range from 0.58 to 1 and from 0.989 to 1.024 on the AMD and NVIDA GPUs, respectively. For connected components, the ratios range from 0.659 to 0.986 and from 0.868 to 1.16 on the AMD and NVIDA GPUs, respectively. Hence, the SYCL applications have not fully achieved performance portability on the GPUs.

The causes of the performance gap are generally attributed to the implementations (e.g., code generation and optimization) of the SYCL compiler for the target GPUs. Particularly, more optimizations are needed for the SYCL compiler with HIP support. Table 10 lists the codes lengths in bytes and register usage of the three compute kernels

Table 8: Performance of the graph coloring applications in CUDA, HIP, and SYCL on the GPUs

Graph name	Mnodes/s (HIP)	Medges/s (HIP)	Mnodes/s (SYCL-HIP)	Medges/s (SYCL-HIP)	Mnodes/s (CUDA)	Medges/s (CUDA)	Mnodes/s (SYCL-CUDA)	Medges/s (SYCL-CUDA)
2d-2d20.sym	2883.9	11524.33	2669.97	10669.45	1106.22	4420.55	1195.27	4776.44
amazon0601	581.31	7042.08	463.84	5619.04	497.2	6023.26	509.56	6172.91
as-skitter	230.63	3016.81	157.1	2290.43	171.53	2243.78	173.26	2266.37
citationCiteseer	724.93	6245.83	522.12	4498.47	951.16	8194.97	948.64	8173.27
cit-Patents	649.85	5687.68	641.47	5614.29	223.92	1959.83	221.62	1939.7
coPapersDBLP	31.34	1767.8	21.71	1224.78	25.32	1428.32	25.11	1416.45
delaunay_n24	1389.05	8334.28	1389.05	8334.29	359.62	2157.73	366.35	2198.09
europe_osm	2694.26	5721.15	2690.72	5713.62	933.06	1981.31	960.64	2039.87
in-2004	58.53	1150.48	37.32	733.57	57.38	1127.9	57.65	1133.11
internet	1393.67	4329.58	808.18	2510.67	2177.31	6764.01	2217.26	6888.13
kron_g500-logn21	14.76	1281.45	14.49	1257.91	26.38	2290.51	26.57	2306.65
r4-2e23.sym	645.54	5164.3	643.73	5149.83	209.18	1673.45	208.59	1668.72
rmat16.sym	130	1919.88	75.44	1114.13	141.5	2089.71	142.08	2098.34
rmat22.sym	149.67	2343.01	150.23	2351.78	83.45	1306.46	83.48	1306.88
soc-LiveJournal1	155.96	2757.23	136.54	2413.95	125.96	2226.9	125.4	2217
uk-2002	108.87	3077.72	83.23	2352.89	93.66	2647.81	93.18	2634.33
USA-road-d.NY	3948.47	10905.33	2850.56	7872.99	6593.38	18210.31	7045.48	19458.99
USA-road-d.USA	3035.05	7313.91	3003.55	7237.99	894.38	2155.28	913.58	2201.55

Table 9: Performance of the connected components application in CUDA, HIP, and SYCL on the GPUs

Graph name	Mnodes/s (HIP)	Medges/s (HIP)	Mnodes/s (SYCL-HIP)	Medges/s (SYCL-HIP)	Mnodes/s (CUDA)	Medges/s (CUDA)	Mnodes/s (SYCL-CUDA)	Medges/s (SYCL-CUDA)
2d-2d20.sym	3773.03	15077.38	3169.33	12664.94	2471.51	9876.4	2332.1	9319.29
amazon0601	1712.25	20742.66	1504.46	18225.37	1335.86	16182.99	1287.96	15602.63
as-skitter	3852.72	37316.09	2548.08	33331.15	1779.5	23277.41	1677.9	21948.39
citationCiteseer	1562.21	13459.67	1222.43	10532.15	1231.12	10607.08	1091.31	9402.51
cit-Patents	1902.16	16648.25	1876.86	16426.8	906.45	7933.5	897.46	7854.83
coPapersDBLP	875.68	49395.65	706.98	39884.3	970.66	54759.62	944.43	53280.14
delaunay_n24	3477.98	20862.44	2987.05	17922.26	950.01	5700.04	904.04	5424.21
europe_osm	4425.53	9397.4	4160.08	8833.73	2564.98	5446.6	2467.13	5238.83
in-2004	2101.03	41298.7	1765.72	34707.59	1919.93	37738.78	1838.04	36129.26
internet	1472.94	4575.84	986.34	3064.17	2444.75	7594.83	2327.77	7231.45
kron_g500-logn21	422.26	36662.15	398.87	34631.58	471.06	40898.88	546.64	47460.68
r4-2e23.sym	2060.04	16480.29	2210.73	17685.79	898.37	7186.96	840.75	6725.97
rmat16.sym	673.25	9942.92	444.07	6558.2	815.04	12036.92	707.47	10448.17
rmat22.sym	1333.96	20882.82	1261.61	19750.24	785.78	12301.13	783.28	12262.02
soc-LiveJournal1	1519.5	26863.95	1360.12	24046.25	972.62	17195.48	936.84	16562.79
uk-2002	1061.35	30004	926.16	26182.67	980.68	27723.75	954.64	26987.6
USA-road-d.NY	2801.07	7736.29	1780.75	4918.28	4577.13	12641.63	4211.25	11631.1
USA-road-d.USA	5679.81	13687.27	4996.2	12039.9	1533.04	3694.33	1483.14	3574.09

Table 10: Code lengths and register usage of the HIP and SYCL kernels (K1, K2, K3) in graph coloring on the AMD GPU

	K1-HIP	K1-SYCL	K2-HIP	K2-SYCL	K3-HIP	K3-SYCL
Code length	1756	2840	2096	4152	772	784
Scalar registers	32	46	33	59	16	23
Vector registers	23	41	37	49	16	14

in HIP and SYCL in graph coloring on the AMD GPU. The code lengths of the first two SYCL kernels are 61.7% and 98.1% longer than those of the HIP kernels, respectively. In addition, the first two SYCL kernels need 43.8% and 78.8% more scalar registers, and 78.3% and 32.4% more vector registers, respectively.

We try to understand causes of the gap in resource usage by analyzing GPU assembly instructions generated by the HIP and SYCL compilers. Our code analysis indicates that a SYCL group function would generate significantly

```
const int thread = threadIdx.x + blockIdx.x * blockDim.x;
const int threads = gridDim.x * blockDim.x
for (int v = thread; __any(v < nodes); v += threads) {
   … …
}
```

Listing 1: Code snippet in HIP for analyzing the assembly instructions generated for the function "__any()"

more assembly instructions, leading to longer code length and higher register utilization. Listings 1 and 2 show the code snippets in HIP and SYCL, respectively. For the HIP group function "__any", the HIP compiler would generate four instructions whereas the SYCL compiler would generate over 60 instructions for the SYCL group function "sycl::any_of_group". The instruction stream generated by the SYCL compiler consists of a sequence of conditional masking on each thread (v_cndmask_b32) and lane permutation (ds_bpermute_b32) instructions. Hence, it is possible to optimize code generation for the SYCL group functions in the SYCL compiler.

```
const int thread = item.get_global_id(0);
const int threads = item.get_group_range(0) * item.get_local_range(0);
for (int v = thread; sycl::any_of_group(sg, v < nodes); v += threads) {
   … …
}
```

Listing 2: Code snippet in SYCL for analyzing the assembly instructions generated for the function sycl::any_of_group()

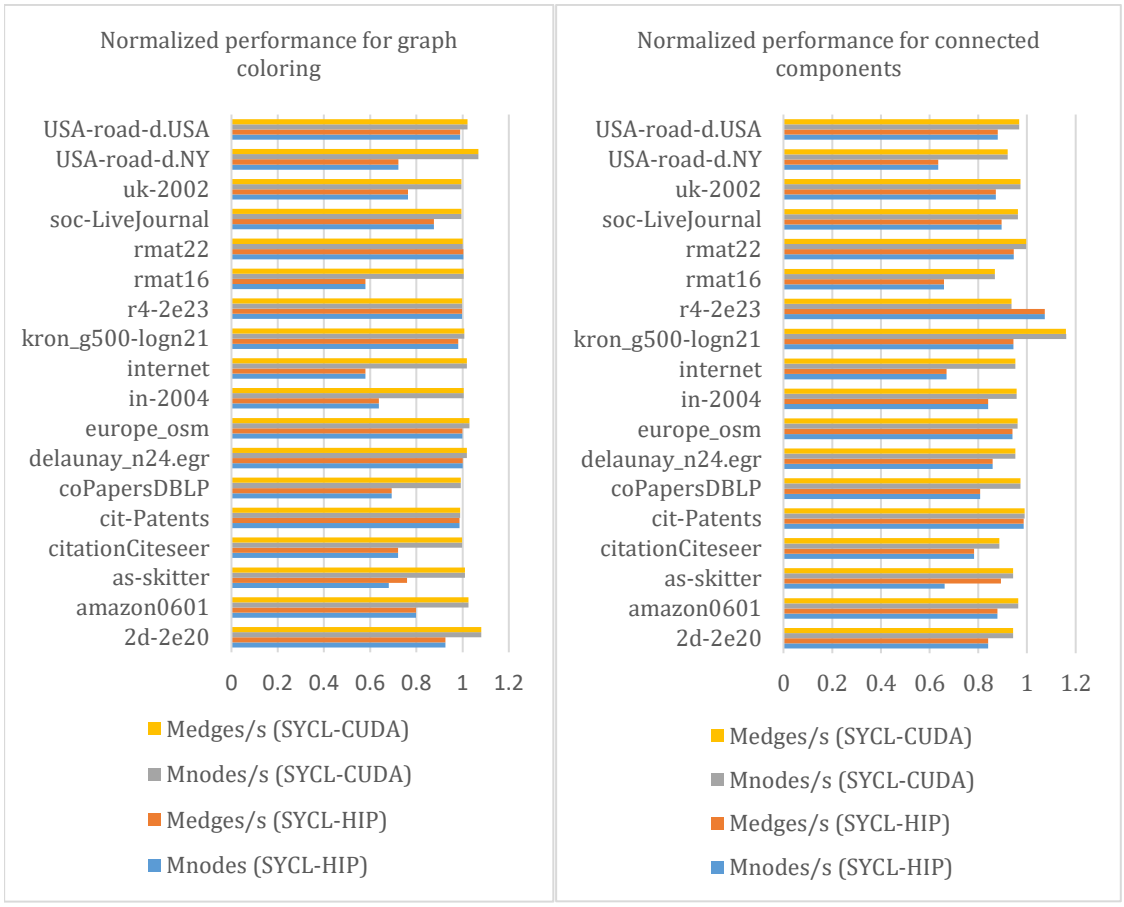

Figure 2. Normalized performance of the SYCL applications on the AMD and NVIDIA GPUs (higher is better)

4.2 Improve the performance of the applications on the GPUs with device-level parallelism

Compiler optimizations are often evasive for most users who have little knowledge of the compiler implementations. Hence, we try an alternative path to improving the performance of the applications. Since the computing platform contains two GPUs, we may reduce the total kernel execution time by executing the compute kernels on both GPUs simultaneously. We assume that this hybrid method could exploit parallel execution of the kernels at the device level. In the modified SYCL program, a SYCL queue is instantiated for each device. We allocate device memory needed by the kernels for each GPU, copy data from the host to the devices, submit kernels of the application to each queue asynchronously, and finally wait for them to complete.

Tables 11 and 12 list the performance of the two SYCL applications running on both GPUs, respectively. For the first two columns in the tables, we compare the performance results of the SYCL applications from Tables 8 and 9 to compute the maximum of the two values. We define "speedup" as the ratio of the performance of the application running on both GPUs over the maximum performance achieved on one of the two GPUs. The speedup ranges from 0.56 to 1.99 for the graph coloring and from 0.6 to 1.897 for connected components. For each application, we find that the speedup results are correlated to the raw performance of the application on the two GPUs. When the kernel execution time of an application on one GPU is about two times faster or slower than that on the other GPU, there is no performance gain from device-level parallelism. In other words, parallelism at the device level is most effective when the execution time of an application on the AMD GPU is close to that on the NVIDIA GPU.

Table 11: Performance of graph coloring in SYCL when executed on the two GPUs

Graph name	Mnodes/s (SYCL-Single)	Medges/s (SYCL-Single)	Mnodes/s (SYCL-Hybrid)	Medges/s (SYCL-Hybrid)	Speedup
2d-2d20.sym	2669.97	10669.45	2175.35	8692.91	0.81
amazon0601	509.56	6172.91	898.38	10883.21	1.76
as-skitter	173.26	2290.43	344.84	4510.87	1.99
citationCiteseer	948.64	8173.27	985.85	8493.88	1.03
cit-Patents	641.47	5614.29	441.41	3863.37	0.68
coPapersDBLP	25.11	1416.45	43.26	2440.61	1.72
delaunay_n24	1389.05	8334.29	726.87	4361.19	0.52
europe_osm	2690.72	5713.62	1886.95	4006.86	0.70
in-2004	57.65	1133.11	74.73	1468.96	1.29
internet	2217.26	6888.13	1317.39	4092.59	0.59
kron_g500-logn21	26.57	2306.65	29.05	2522.25	1.09
r4-2e23.sym	643.73	5149.83	416.05	3328.36	0.64
rmat16.sym	142.08	2098.34	141.85	2094.96	0.99
rmat22.sym	150.23	2351.78	165.36	2588.63	1.10
soc-LiveJournal1	136.54	2413.95	252.71	4467.71	1.85
uk-2002	93.18	2634.33	167.27	4728.77	1.79
USA-road-d.NY	7045.48	19458.99	3964.29	10949	0.56
USA-road-d.USA	3003.55	7237.99	1802.09	4342.7	0.59

Table 12: Performance of connected components in SYCL when executed on the two GPUs

Graph name	Mnodes/s (SYCL - Single)	Medges/s (SYCL - Single)	Mnodes/s (SYCL- Hybrid)	Medges/s (SYCL- Hybrid)	Speedup
2d-2d20.sym	3169.33	12664.94	3803.64	15199.71	1.20
amazon0601	1504.46	18225.37	2181.56	26427.99	1.45
as-skitter	2548.08	33331.15	2929.56	38321.21	1.14
citationCiteseer	1222.43	10532.15	1526.96	13155.92	1.24
cit-Patents	1876.86	16426.80	1524.27	13340.83	0.81
coPapersDBLP	944.43	53280.14	1276.96	72039.76	1.35
delaunay_n24	2987.05	17922.26	1792.33	10753.94	0.60
europe_osm	4160.08	8833.73	4913.33	10433.22	1.18
in-2004	1838.04	36129.26	2814.33	55319.55	1.53
internet	2327.77	7231.45	1485.99	4616.35	0.63
kron_g500-logn21	546.64	47460.68	806.64	70034.8	1.47
r4-2e23.sym	2210.73	17685.79	1670.57	13364.57	0.75
rmat16.sym	707.47	10448.17	645.46	9532.48	0.91
rmat22.sym	1261.61	19750.24	1374.4	21515.93	1.08
soc-LiveJournal1	1360.12	24046.25	1774.22	31367.20	1.30
uk-2002	954.64	26987.60	1811.88	51221.97	1.89
USA-road-d.NY	4211.25	11631.10	2780.85	7680.46	0.66
USA-road-d.USA	4996.20	12039.90	2947.34	7102.54	0.58

5 RELATED WORK

Many studies have focused on performance and portability of SYCL on vendors' computing platforms. In [35], the authors evaluate the performance of benchmarks and mini-apps having both SYCL and CUDA implementations on an NVIDIA Volta GPU. They conclude that the performance of running SYCL can be competitive with using CUDA directly. In [36], 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 migrating 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. In [37], 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 [38], 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 [39], 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 [40], 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 in the application programming interfaces between CUDA and SYCL. The optimizations applied by a compiler to a kernel also pose challenges and complexities to performance portability. Compared to the findings in [41], more SYCL extensions have been added to facilitate the CUDA migration process. In addition, we extend the work with performance evaluation of two graph applications on the NVIDIA and AMD GPUs and explore performance optimization with device-level parallelism.

The CUDA-to-SYCL conversion tool can automate the migration process by automatically generating variants of SYCL codes [42]. This significantly improves productivity compared to manual conversion. However, the generated codes often require manual changes in GPU kernels for performance [43, 44, 45], and certain experimental features, such as cache configuration, are not supported yet.

6 CONCLUSION

The plugin interfaces in the SYCL compiler facilitate functional portability of a SYCL program across vendors' computing platforms. While programming models serve the same purpose of accelerating applications on GPUs, a good understanding of vendors' programming models is still needed for migrating applications from CUDA to SYCL. Comparing the performance of the applications in CUDA, HIP, and SYCL calls for improving code optimizations and register usage in the compiler implementation for performance portability. Exploiting device-level parallelism requires manual changes of the SYCL program, and the performance speedup depends on the performance of the application on each device. With the development of the SYCL compilers and applications from the community, we hope our findings will help improve functional and performance portability.

ACKNOWLEDGMENT

We appreciate the reviewers for their comments and suggestions. This research used resources of the Experimental Computing Lab at Oak Ridge National Laboratory. 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] SYCL Extensions in DPC++. [online] https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/

[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] Wu, M., Li, X., Kwoh, C.K. and Ng, S.K., 2009. A core-attachment based method to detect protein complexes in PPI networks. BMC bioinformatics, 10(1), pp.1-16.

[15] Hossam, M.M., Hassanien, A.E. and Shoman, M., 2010, November. 3D brain tumor segmentation scheme using K-mean clustering and connected component labeling algorithms. In 2010 10th International Conference on Intelligent Systems Design and Applications (pp. 320-324). IEEE.

[16] He, L., Ren, X., Gao, Q., Zhao, X., Yao, B. and Chao, Y., 2017. The connected-component labeling problem: A review of state-of-the-art algorithms. Pattern Recognition, 70, pp.25-43.

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

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

[19] 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.

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

[21] 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).

[22] 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).

[23] Di Stefano, L. and Bulgarelli, A., 1999, September. A simple and efficient connected components labeling algorithm. In Proceedings 10th international conference on image analysis and processing (pp. 322-327). IEEE.

[24] Azami, N. and Burtscher, M., 2022, November. Compressed In-memory Graphs for Accelerating GPU-based Analytics. In 2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3) (pp. 32-40). IEEE.

[25] Jaiganesh, J. and Burtscher, M., 2018, June. A high-performance connected components implementation for GPUs. In Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (pp. 92-104).

[26] 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).

[27] 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.

[28] The Intel DPC++ compiler. https://github.com/intel/llvm

[29] 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.

[30] 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).

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

[32] Radeon Open Compute (ROCm) Platform. https://rocmdocs.amd.com

[33] https://github.com/intel/llvm-test-suite/pull/1687

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

[35] 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).

[36] 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.

[37] 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).

[38] 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.

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

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

[41] Jin, Z., 2022. Experience of Migrating Parallel Graph Coloring from CUDA to SYCL (No. ORNL/TM-2022/2433). Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States).

[42] Huang, A., 2023, April. SYCLomatic compatibility library: making migration to SYCL easier. In Proceedings of the 2023 International Workshop on OpenCL (pp. 1-2).

[43] 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.

[44] Castaño, G., Faqir-Rhazoui, Y., García, C. and Prieto-Matías, M., 2022. Evaluation of Intel's DPC++ Compatibility Tool in heterogeneous computing. Journal of Parallel and Distributed Computing, 165, pp.120-129.

[45] Tsai, Y.H.M., Cojean, T. and Anzt, H., 2022. Providing performance portable numerics for Intel GPUs. Concurrency and Computation: Practice and Experience, p.e7400