ANL/ALCF-19/4

Improving the Performance of Medical Imaging Applications using SYCL

Argonne Leadership Computing Facility

ANL/ALCF-19/4

Improving the Performance of Medical Imaging Applications using SYCL

prepared by Zheming Jin

Argonne Leadership Computing Facility, Argonne National Laboratory

December 3, 2019

Improving the Performance of Medical Imaging Applications using SYCL

I. Introduction

In this report, we are interested in applying the SYCL programming model to medical imaging applications for a study on performance portability and programming productivity. The SYCL standard specifies a cross-platform abstraction layer that enables programming of heterogeneous computing systems using standard C++. As opposed to the Open Computing Language (OpenCL) programming model, in which host and device code are generally written in different programming languages [1], SYCL can combine host and device code for an application in a type-safe way to improve development productivity and performance portability.

Rodinia is a widely used open-source benchmark suite for heterogeneous computing. We choose two medical imaging applications (Heart Wall and Particle Filter) in the Rodinia benchmark suite [2], migrate the OpenCL implementations of the applications to the SYCL implementations, and evaluate their performance and productivity on Intel® microprocessors that contains a central processing unit (CPU) and an integrated graphics processing unit (GPU). Currently, the maturing SYCL compilers, based on a conformant implementation of the SYCL 1.2.1 Khronos specification, are optimized for Intel® computing platforms.

The experimental results are promising in terms of the raw performance and productivity. Although the SYCL implementation of the Heart Wall application does not execute successfully on a CPU, it is on average 15% faster than the OpenCL implementation on an Intel® Iris™ Pro integrated GPU. For the Particle Filter application, the performance difference between the SYCL and OpenCL implementations is less than 1% on the GPUs for most cases, but the SYCL implementation is on average 4.5X faster on an Intel® Xeon® four-core CPU. Arguably, we use lines of code as a way to measure programming productivity in software. The SYCL programs reduce the lines of code of the OpenCL programs by 52% and 38% for the Heart Wall and Particle Filter, respectively. The results indicate that SYCL is a promising programming model for heterogeneous computing with the maturing compilers.

We organize the remainder of the report as follows. Section II introduces the SYCL programming model, compares the major differences between an OpenCL program and a SYCL program, and describes the characteristics of the two applications. Section III describes the SYCL programming model in more details, and shows the SYCL implementation of a kernel function in the Particle Filter as an example. In Section IV, we evaluate and analyze the performance of the applications on the CPUs and GPUs. Section V concludes the report.

II. Background

A. *SYCL*

C++ AMP, CUDA, Thrust C++ are representative single-source C++ programming models for accelerators [3]. Such languages are easy to use and can be type-checked as everything sits in a single source file. They facilitate offline compilation so that the binary can be checked at compile time. A SYCL program, based on a single-source C++ model as shown in Figure 1, can be compiled for a host while kernel(s) are extracted from the source and compiled for a device. A SYCL device compiler parses a SYCL application and generates intermediate representations (IR). A standard host compiler parses the same application to generate native host code. The SYCL runtime will load IR at runtime, enabling other compilers to parse it into native device code. Hence, people can continue to use existing toolchains for a host platform, and choose preferred device compilers for a target platform.

The design of SYCL allows for the combination of the performance and portability features of OpenCL and the flexibility of using high-level C++ abstractions. Most of the abstraction features of C++, such as templates, classes, and operator overloading, are available for a kernel function in SYCL. Some C++ language features, such as virtual functions, virtual inheritance, throwing/catching exceptions, and run-time type-information, are not allowed inside kernels due to the capabilities of the underlying OpenCL standard. These features are available outside kernel scope.

A SYCL application is logically structured in three scopes: application scope, command-group scope, and kernel scope. The kernel scope specifies a single-kernel function that will be executed on a device after compilation. The command-group scope specifies a unit of work that will comprise of a kernel function and buffer accessors. The

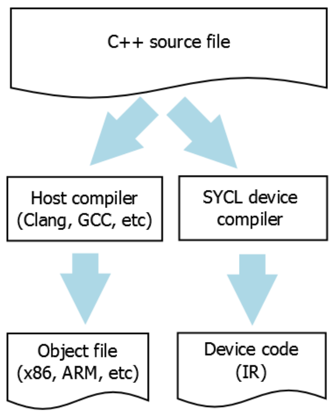


Figure 1. SYCL is a single-source programming model

TABLE I. MAPPING FROM OPENCL TO SYCL

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

application scope specifies all other codes outside of a command-group scope. A SYCL kernel function may be defined by the body of a lambda function, by a function object or by the binary generated from an OpenCL kernel string. Although an OpenCL kernel is interoperable in the SYCL programming model, in this report we focus on the implementation of kernel functions using lambda functions.

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

B. *Medical Imaging Applications in Rodinia*

Rodinia is a widely used open-source benchmark suite for heterogeneous computing. From the suite we choose two medical imaging applications. Particle Filter (PF) is a medical imaging application for tracking leukocytes and myocardial cells [4]. The algorithm may be used in other domains such as video surveillance and video compression. Heart Wall (HW) is another medical imaging application that tracks the movement of a mouse heart over numerous ultrasound images [5]. Both applications support OpenCL and CUDA programming models for a heterogeneous computing device.

TABLE II. CHARACTERISTICS OF THE TWO OPENCL APPLICATIONS

App.	LOC (host)	LOC (kernel)	Number of kernels	Max number of kernel arguments
HW	775	1043	1	34
PF	720	149	4	20

Table II summarizes the characteristics of the two applications in terms of the OpenCL host and kernel programs. For the PF application, we focus on the single-precision floating-point version. We measure LOC of the OpenCL host and kernel programs using the utility "cloc" [6], and count the number of kernels and the maximum number of kernel arguments of a single or multiple kernel(s). While comments and blank lines are not included, and multiple lines are joined into one line for the OpenCL built-in function calls, the LOC of both applications indicate that developing the host programs is tedious and error-prone. In the original OpenCL and CUDA applications, the authors reduced the significant GPU overhead of data transfer and kernel launch for the kernel of the HW application by combining multiple kernels into one kernel call. Hence, the kernel size of the HW application is much larger than the sizes of the four kernels in the PF application. On the other hand, both kernels require a great number of kernel arguments. They reduce programming productivity as it makes debugging difficult when the order of setting kernel arguments in a host program does not strictly match that in a kernel function.

III. IMPLEMENTATIONS

Overall, it is relatively straightforward to port an OpenCL application to a SYCL application by following the steps listed in Table I. However, we would like to describe the experience we learn from our porting efforts.

A. *Buffer Construction*

A SYCL buffer differs from an OpenCL buffer in that it can handle both storage and ownership of data. In addition, a buffer is destroyed when it goes out of scope. Table III lists the ways a buffer can be constructed and its initial values after construction. The destruction behavior indicates if the SYCL runtime will block until all work in queues on the buffer have completed. In our experiment, we can just use the first two methods for constructing buffers, which closely match the OpenCL buffer management in the

TABLE III. SUMMARY OF SYCL BUFFER MANAGEMENT

Construction Method	Buffer Content after Construction	Destruction Behavior
Buffer size	Uninitialized	Non-blocking
Associated host memory	Contents of host memory	Blocking
Unique pointer to host data	Contents of host data	Blocking
Shared pointer to host data	Contents of host data	Blocking
A pair of iterator values	Data from the range defined by the iterator pair	Non-blocking
OpenCL memory object	OpenCL memory object	Blocking

TABLE IV. SYCL BUFFER ACCESS MODES

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

applications. Although it is not necessary to use either a unique pointer or a shared pointer to host data, we will look at the two methods, which does not necessarily require data copy-back from device to host, in our future work. Using a pair of iterator values for a range of values is flexible, but it is not needed for the two application under study. The last method, which allows for the interoperability of OpenCL and SYCL, is not our focus.

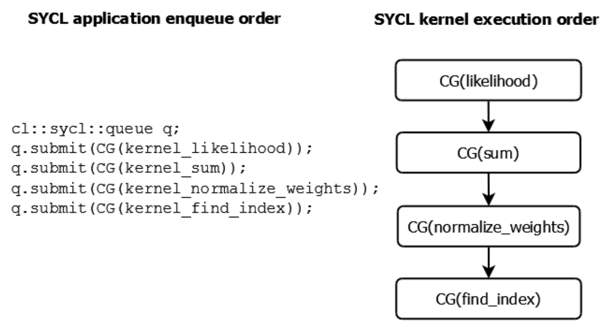
B. *Buffer Access Mode*

SYCL accessors allow a user to specify the types of data access, and the SYCL implementation ensures that the data is accessed appropriately. A device accessor, which is the default access type, allows a kernel to access data on a device. In contrast, a host accessor gives access to data on the host. A device accessor can only be constructed within command groups whereas a host accessor can be created outside command groups. Constructing a host accessor is blocking by waiting for all previous operations on the underlying buffer to complete.

An accessor must be specified with an access mode shown in Table IV. Discarding write indicates that previous contents of a device buffer is not preserved, which implies that it is not necessary to copy data from host to device before the buffer is accessed. It is important to specify the access mode correctly; otherwise, the compiler will report an error when a kernel function tries to write to a read-only buffer. On the other hand, a read-only accessor to a buffer can disable data copy to host memory when the buffer is destroyed.

C. *Data Movement between Host and Device*

For OpenCL applications, data transfers between a host and a device are explicitly made with the OpenCL built-in functions "clEnqueueReadBuffer()" and "clEnqueueWriteBuffer()". In SYCL, we can rely on implicit data transfers realized by buffer accessors. When a buffer is constructed with associated host memory as shown in Table III, the pointer to the host memory allows SYCL runtime to copy data back from device to host before the buffer is destroyed. Without explicit data copy specified in a SYCL program, the SYCL compiler will generate OpenCL built-in functions "clEnqueueMapBuffer()" and "clEnqueueUnmapMemObject()" for moving data between host and device. However, we should not use the pointer on the host side to access the contents of the buffer before the buffer is destroyed. Instead, we can use a host accessor to access the memory managed by a SYCL buffer.



```
cl::sycl::queue q;
q.submit(CG(kernel_likelihood));
q.submit(CG(kernel_sum));
q.submit(CG(kernel_normalize_weights));
q.submit(CG(kernel_find_index));
```

Figure 2. Execution order of the four SYCL kernels in PF

D. *Kernel Execution Order*

In OpenCL, a command queue is required to transfer data between a host and a device, and to ensure different kernels execute in the correct order. In contrast, SYCL provides an abstraction that only requires users to specify which data are needed to execute a kernel. By specifying access modes and types of memory, a directed acyclic dependency graph of different kernels is constructed at runtime based on the relative order of command-groups submissions to a queue.

For example, in the PF application there are four kernels which are executed in each image frame. After they are submitted to a queue, a dependency graph is built as shown in Figure 2. While each command group (CG) for a kernel is submitted to a SYCL queue asynchronously, the runtime will determine the execution order of these kernels based on the read and write access modes of device accessors.

E. *Kernel Execution Model*

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

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

Listing 1 presents the SYCL implementation of the "normalize_weights" kernel in the PF application under the scope of command group. We manually inline the function calls in the original OpenCL kernel. For the kernel, we focus on the semantics of the program rather than how weights are normalized. The operations, which process data on a device, are represented using a command group function object. The function object is given a command group handler (cgh) object to perform all the necessary work required to process data on a device using a kernel. The group of commands for data transferring and processing is enqueued as a command group on a device. A command group is submitted to a SYCL command queue for execution. Accesses to the buffers are controlled via device accessors constructed through the "get_access" method of the buffers (lines 2-7). For simplicity, we omit the specifications of the buffer access modes for all the accessors. We also construct two local accessors (lines 8-9), which provide accesses to the allocated shared memories on a device. Each local memory is shared among all work-items of a work-group. The memory allocated by a local accessor is not initialized, so it is reset by the first work-item in a work-group (line 14). In the kernel function, the identifiers of global and local work-items are retrieved with the member functions of the "nd_item" class (lines 11-12). The barrier function is required to synchronize all work-items in a work-group in local or global memory space (lines 16, 20, 32, 36). The floating-point math functions (i.e., *fabs*, *sqrt*, and *cos*) need to be qualified in the SYCL namespace (lines 27, 29, 30) to tell a SYCL compiler that these math functions, which are not confused with the math functions called on a host, will be executed on a device.

IV. Experiment

A. Setup

We use the Codeplay SYCL compiler (ComputeCpp™ community edition, version 1.1.6) as we find that the compiler is more mature than the open-source SYCL compiler [7] for the two applications. We choose two server platforms in our experiment. One server has an Intel® Xeon® E3-1284L v4 CPU running at 2.9 GHz. The CPU has four cores and each core supports two threads. The integrated GPU is Broadwell GT3e, Generation 8.0. It contains 48 execution units, and each execution unit corresponds to a compute unit in the OpenCL programming model. The maximum dynamic frequency of the GPU is 1.15 GHz. The other server has an Intel® Xeon® E3-1585 v5 CPU running at 3.5 GHz. The CPU also has four cores and each core supports two threads. The integrated GPU is Skylake GT3e, Generation 9.0. It contains 72 execution units. The maximum dynamic frequency of the GPU is 1.15 GHz. A few details of the two GPUs are listed in Table V.

For the GPU compute runtime, the device version is OpenCL 2.1 NEO and the driver version 19.43.14583. The maximum work-group size on a GPU is 256. For the CPU runtime, the device versions is OpenCL 2.1. The maximum work-group size is 8192. The operating system on the server with the Broadwell microprocessor is CentOS Linux 7 and the kernel 5.3.10. The operating system on the server with the Skylake microprocessor is Red Hat Enterprise Linux 7 and the kernel 5.3.1.

We compile the SYCL programs with the options "-O3 -no-serial-memop -sycl-driver", and the OpenCL programs using the options "-O3" and the GNU compiler, version 4.8.5. We evaluate the performance of the two applications with the test scripts provided by the benchmark suite. We

```
 1 q.submit([&] (handle& cgh) {
 2   auto weights = d_weights.get_access<read_write>(cgh);
 3   auto Nparticles = d_Nparticls.get_access<…>(cgh);
 4   auto partial_sums = d_partial_sums.get_access<…>(cgh);
 5   auto CDF = d_CDF.get_access<…>(cgh);
 6   auto u = d_u.get_access<…>(cgh);
 7   auto seed = d_seed.get_access<…>(cgh);
 8   accessor<float,1,read_write,access::target::local>
        u1(1,cgh);
 9   accessor<float,1,read_write,access::target::local>
        sumWeights(1,cgh);
10   cgh.parallel_for<class normalize_weights>(
        nd_range<1>(range<1>(global_work_size),
                    range<1>(local_work_size)),
                    [=](nd_item<1> item) {
11     int i = item.get_global_id(0);
12     int lid = item.get_local_id(0);
13     if (lid == 0) {
14       sumWeights[0] = partial_sums[0];
15     }
16     item.barrier(access::fence_space::local_space);
17     if (i < Nparticles[0]) {
18       weights[i] = weights[i]/sumWeights[0];
19     }
20     item.barrier(access::fence_space::global_space);
21     if (i == 0) {
22       CDF[0] = weights[0];
23       for(int x = 1; x < Nparticles[0]; x++){
24         CDF[x] = weights[x] + CDF[x-1];
25       }
26       seed[i] = (A*seed[i] + C) % M;
27       float p = cl::sycl::fabs(seed[i]/((float)M));
28       seed[i] = (A*seed[i] + C) % M;
29       float q = cl::sycl::fabs(seed[i]/((float)M));
30       u[0]    = (1/((float)(Nparticles[0]))) *
                  (cl::sycl::sqrt(-2*cl::sycl::log(p))*
                   cl::sycl::cos(2*PI*q));
31     }
32     item.barrier(access::fence_space::global_space);
33     if (0 == local_id) {
34       u1[0] = u[0];
35     }
36     item.barrier(access::fence_space::local_space);
37     if (i < Nparticles[0]) {
38       u[i] = u1[0] + i/((float)(Nparticles[0]));
39     }
40   });
41 });
```

Listing 1. Implementation of the "normalize_weights" kernel in SYCL

TABLE V. Summary of the two Integrated GPUs

Parameter	Iris™ Pro Graphics P580	Iris™ Pro Graphics P6300
Generation	Gen9 (Skylake)	Gen8 (Broadwell)
Technology	14 nm	14 nm
Base Freq.	0.35 GHz	0.3 GHz
Max Dynamic Freq.	1.15 GHz	1.15 GHz
Embedded DRAM	128 MB	128 MB
Slices/Subslices	3/9	2/6
Execution Units	72	48
Max GFLOPS	1325	883

choose the elapsed time of executing the entire application as our performance metric. For both applications, the offloading time consumes approximately 99% of the entire application time. When measuring the execution time of the OpenCL implementations, we disable checking the error status after invoking each OpenCL built-in function to remove its effect on the timing.

B. *Experimental Results*

The execution time in second of running the HW implemented in SYCL and OpenCL on the P6300 GPU and P580 GPU is displayed in Figures 3 and 4, respectively. Increasing the work-group size is effective in improving the performance of the application. Because the execution time is even longer for work-group sizes smaller than 32, they are not shown in the figures. Based on the profiling results of the SYCL and OpenCL implementations on the GPUs, we attribute the shorter execution time of the SYCL implementations to the shorter kernel execution time on the on the two GPUs. The SYCL compiler also reduces the overhead of data transfers between the host and device although the kernel execution consumes 99% of the device time. On the other hand, the execution time of the SYCL and OpenCL implementations are almost the same for certain work-group sizes on the two GPUs. Since the performance of the application is intimately associated with the work-group size, the results suggest that the SYCL runtime is more efficient in scheduling the execution of work-groups on each GPU for certain work-group sizes. Currently, the binaries generated from the SYCL programs does not run successfully on the CPUs. Hence, the comparison of the SYCL and OpenCL implementations on the CPUs is not available.

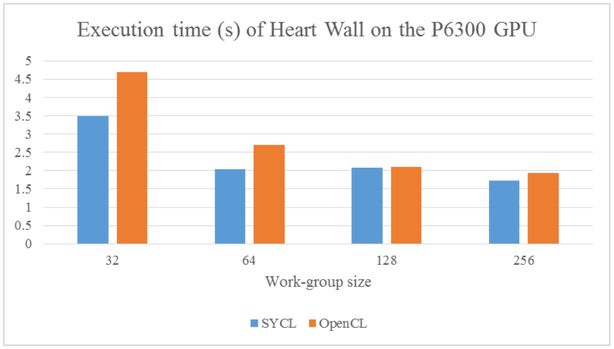


Figure 3. Performance comparison of the SYCL and OpenCL implementations of Heart Wall across the work-group sizes on the P6300 GPU

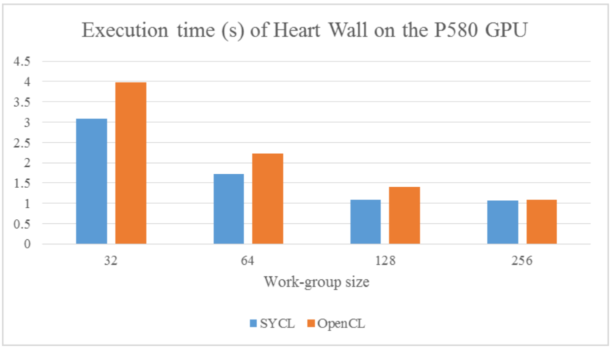


Figure 4. Performance comparison of the SYCL and OpenCL implementations of Heart Wall across the work-group sizes on the P580 GPU

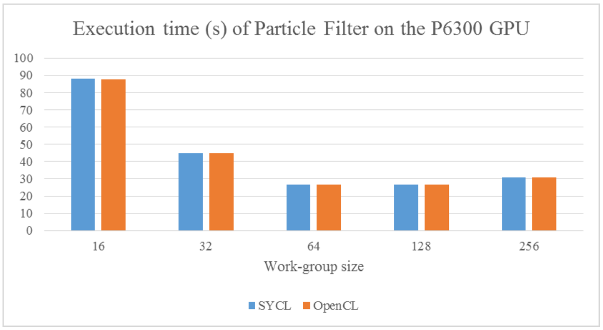


Figure 5. Performance comparison of the SYCL and OpenCL implementations of Particle Filter across the work-group sizes on the P6300 GPU

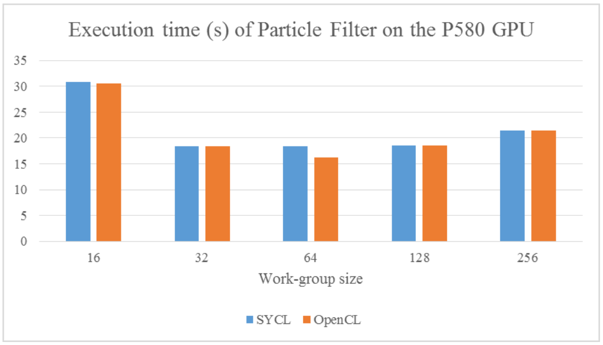


Figure 6. Performance comparison of the SYCL and OpenCL implementations of Particle Filter across the work-group sizes on the P580 GPU

Figures 5 and 6 show the execution time in second of running the PF in SYCL and OpenCL on the two GPUs, respectively. There is a sweet spot where the highest performance on each GPU is achieved using a work-group size of 64. As the execution time is even longer for work-group sizes smaller than 16, they are not shown in the figures. The difference in execution time between the SYCL and the OpenCL implementations is less than 1% for almost all cases.

As shown in Figures 7 and 8, on the CPUs the highest performance levels off when the work-group size is less than 64. The results of device profiling indicate that the significant slowdown in the performance of the OpenCL implementation is caused by the OpenCL likelihood kernel. The OpenCL kernel is 11X slower than the SYCL kernel.

In terms of programming productivity, we argue that the productivity is often associated with the size of a program in terms of lines of code (LOC). The LOC of the HW are

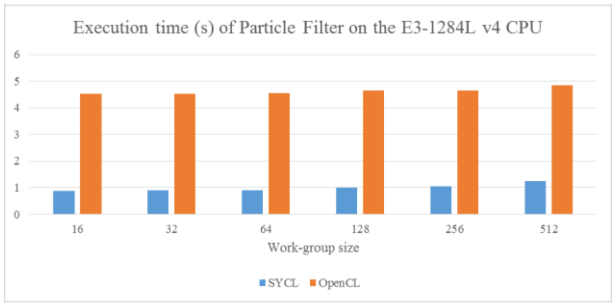


Figure 7. Performance comparison of the SYCL and OpenCL implementations of Particle Filter across the work-group sizes on the E3-1284L v4 CPU

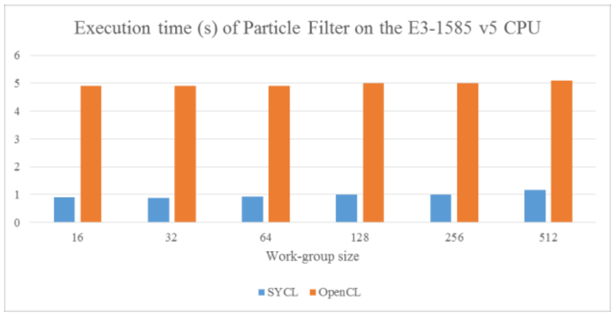


Figure 8. Performance comparison of the SYCL and OpenCL implementations of Particle Filter across work-group sizes on the E3-1585 v5 CPU

approximately 370 and the LOC of the PF approximately 450. Hence, the SYCL program reduces the LOC of the OpenCL program listed in Table II by 52% and 38%, respectively. Given that the SYCL kernels are almost the same as the OpenCL kernels, there are a few factors that contribute to the decrease of LOC. Using a device selector in SYCL greatly simplifies the search for platforms and devices in OpenCL. In addition, a lambda expression in SYCL removes the need to explicitly build a kernel program and set kernel arguments as in OpenCL. For a host program, the improvement of programming productivity is more evident for a kernel with a large number of kernel arguments specified in global memory space. Each kernel argument may require data transfer between host and device. The implicit data transfer between host and device in SYCL also contributes to the decrease in LOC.

V. CONCLUSION

SYCL is a single-source programming model that allows kernel codes to be embedded in host codes. In this report, we apply the SYCL programming model to the medical imaging applications in the open-source Rodinia benchmark suite, describe our experience of transforming the OpenCL implementations to the SYCL implementations, and evaluate their performance on Intel® microprocessors with a CPU and an integrated GPU. While the transformation from OpenCL to SYCL is relatively straightforward given that the SYCL programming model is an extension to OpenCL, understanding buffer accessors, kernel execution order and model, and program scopes is important for the smooth transformation of the applications. The experimental results are promising in terms of the raw performance and the programming productivity which can be achieved using the SYCL programming model. The maturing SYCL compilers will continue to promote performance, portability, and productivity.

ACKNOWLEDGMENT

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

REFERENCES

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

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

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

[4] Goodrum, M.A., Trotter, M.J., Aksel, A., Acton, S.T. and Skadron, K., 2010, June. Parallelization of particle filter algorithms. In International Symposium on Computer Architecture (pp. 139-149). Springer, Berlin, Heidelberg.

[5] Szafaryn, L.G., Skadron, K. and Saucerman, J.J., 2009, June. Experiences accelerating MATLAB systems biology applications. In Proceedings of the Workshop on Biomedicine in Computing: Systems, Architectures, and Circuits (pp. 1-4).

[6] Danial, A., CLOC—Count lines of code, 2009. URL http://cloc.sourceforge.net.

[7] https://github.com/intel/llvm/blob/sycl/sycl/ReleaseNotes.md