Experience Migrating OpenCL to SYCL: A Case Study on Searches for Potential Off-Target Sites of Cas9 RNA-Guided Endonucleases on AMD GPUs

Zheming Jin
Oak Ridge National Laboratory
jinz@ornl.gov

Jeffrey S. Vetter
Oak Ridge National Laboratory
vetter@computer.org

***Abstract*—Cas-OFFinder is a popular application written in OpenCL for searching potential off-target sites in parallel on a GPU. In this work, we describe our experience of migrating the application from OpenCL to SYCL. Evaluating the performance of the OpenCL and SYCL application using human genome sequences shows that the SYCL program could achieve performance portability on the target GPUs. Exploring the optimizations of the hotspot kernel in SYCL may further improve the performance of the application by 9% to 23%.**

Keywords—Programming model, heterogeneous computing, sequence analysis

I. Introduction

Open Computing Language (OpenCL) [1, 2] is an open standard supported by major graphics hardware and personal computer vendors interested in offloading compute intensive workloads (kernels) to heterogeneous computing devices such as graphics processing units (GPUs) for acceleration. OpenCL promotes portability, allowing a program to execute on a variety of computing platforms with the support of OpenCL compilers. However, writing an OpenCL program tends to be error-prone and time-consuming compared to other popular programming models [3, 4]. SYCL is a programming model that builds on the underlying concepts, portability, and efficiency of OpenCL while adding much of the ease of use and flexibility of single-source C++ [5, 6]. SYCL attempts to gain the simplicity of writing a single program and to enable compilers to statically type-check the correctness of the program. When migrating a program from OpenCL to SYCL, it is desirable that the program could still achieve reasonable performance on various platforms. Hence, it is worthwhile to investigate performance and portability we may obtain with SYCL.

In this paper, we choose a bioinformatics application that searches in parallel for potential off-target sites in genome sequences as a case study on performance and portability of SYCL. We convert the application from OpenCL to SYCL and explain the migration paths. Then, we evaluate the performance of the OpenCL and SYCL applications on AMD GPUs and explore optimization techniques for improving the performance of the hotspot kernel in the SYCL program. The experimental results show that the SYCL application could achieve performance portability on the GPUs. Furthermore, exploring the kernel optimizations can further improve the performance of the SYCL program by 9% to 23%. While the application in our case study covers only a subset of the features in the SYCL specification [7], the experiences migrating OpenCL to SYCL in our case study can be applied to other applications.

We have described the motivation and scope of our work. The remainder of the paper is organized as follows. Section II summarizes the bioinformatics application in OpenCL and contrasts the OpenCL and SYCL programming from an application aspect. Section III describes the SYCL application in detail by explaining the migration paths between OpenCL and SYCL. Section IV presents the experimental results including performance evaluation, profiling, and optimization. Section V is a summary of related work. Section VI concludes the paper.

II. Background

A. Summary of Cas-OFFinder

Cas-OFFinder is a fast and versatile algorithm that searches for potential off-target sites of Cas9 RNA-guided endonucleases [8, 9]. The application implements the algorithm with OpenCL for exploiting data parallelism on GPUs. In the application, the OpenCL host program reads genome sequence data in single- or multi-sequence data format, parses the data files with an open-source parser library, and divides the genome data into chunks that can fit the memory of a heterogenous device. Then, these chunks are fed into a "search" kernel to select specific sites that include a protospacer-adjacent motif (PAM) sequence [10]. To search and select these specific sites efficiently, the OpenCL kernel performs parallel search over the sites in a chunk. After the kernel is complete, the OpenCL host program collects the information of the specific sites that contain PAM sequences and sends these sequences to a "compare" kernel in OpenCL. The kernel counts the number of mismatched bases in parallel. After the kernel is complete, the host program selects potential off-target sites that contain mismatched bases under a given threshold, and saves the results (chromosome number, position, direction, the number of mismatched bases and potential off-target DNA sequence with mismatched bases) in a file for analysis. The interaction between the OpenCL host and kernel programs continues until all chunks are processed. As

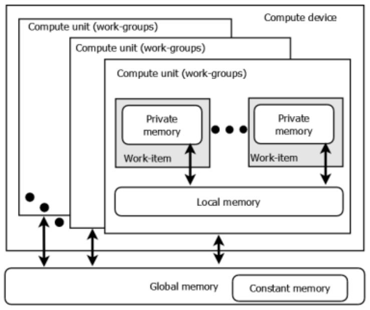

Fig. 1. Abstract memory model

mentioned in [11], Cas-OFFinder is one of the most popular tools for searching potential off-target sites, with no limit to the number of mismatches, PAM types, etc. It can also predict off-target sites with deletions or insertions.

B. *Abstract memory model*

Figure 1 shows an abstract view of a memory hierarchy in which a kernel typically executes. To hide memory access latency and obtain high arithmetic throughput, a kernel is typically executed in multiple instances. Each instance is considered as a work-item for a single element of work. These instances are organized into work-groups that can be indexed in one, two, or three dimensions. The total number of work-items in a work-group is work-group size. Work-groups execute on a compute unit that comprises private memories, processing units and memory interfaces. A device global memory can be accessed by all work-items in work-groups. A constant memory may be utilized to store constant values across work-items in work-groups. A shared local memory enables low-latency communication between work-items within a work-group. A building block for communication is a barrier function. A barrier, which synchronizes execution of work-items in a work-group, ensures that all work-items have finished an operation before using the result of that operation. It also ensures that results of memory operations performed before a barrier can be seen by other work-items after the barrier.

C. *Contrast OpenCL and SYCL programming*

In Table I, we contrast the general steps of writing an OpenCL program and those of a SYCL program. The first three steps in OpenCL are reduced to an instance of the SYCL device selector class. A selector searches a device of a user's provided preference (e.g., GPU) at runtime. The SYCL queue class encapsulates a command queue for offloading kernels to a device. A kernel function in SYCL, which is invoked as a lambda function, is submitted to execution via a command queue. Hence, steps 6 to 10 in OpenCL are reduced to the definition and execution of of a lambda expression and via a SYCL queue. Data transfers between a host and a device may be realized by the SYCL accessor objects and memory copy commands, and the event handling can be handled by the SYCL event class. An OpenCL program needs to release the allocated sources of queue, program, kernel, and memory objects explicitly. In SYCL, they can be handled by the SYCL runtime which implicitly calls destructors.

TABLE I. PROGRAMMING STEPS IN OPENCL AND SYCL

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

The total numbers of logical programming steps are 13 and 8 for the OpenCL and SYCL programs, respectively. Hence, SYCL could improve programming productivity with abstractions, relieving a programmer from the burden of managing device, program, kernel, and memory objects in OpenCL.

III. EXPERIENCES MIGRATING OPENCL TO SYCL

While there exists a comprehensive guide on migrating OpenCL to SYCL codebase from a vendor, this section gives a detailed explanation of the migration paths between OpenCL and SYCL in the application.

A. *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. To migrate the OpenCL program, we get started with SYCL buffers for data management in our study. A SYCL buffer defines a data structure of one, two or three dimensions that can be accessed by a kernel. The underlying data type of a buffer must be trivially copyable as defined by C++. A SYCL buffer is considered as a high-level data abstraction for data management because properties of a buffer can be queried to determine whether and where device data is read from or written back to host memory. However, accessing the underlying data in a buffer requires an SYCL accessor object. Such object indicates where and how data is accessed.

Table II contrasts the memory management using OpenCL and SYCL buffers. For clarity, we will omit the SYCL namespace in the following examples. In OpenCL, a memory object is allocated by creating a memory buffer with a context (ctx), access flags (flags), buffer size in bytes (BS), an optional pointer to a host memory (h), and error status (err). In SYCL, a buffer is instantiated with the specifications of the data type (T), dimension (D), and word size (WS) of the underlying data. The initial content of the buffer is not specified. The constructed SYCL buffer will use a default allocator when allocating memory on a host. A SYCL buffer can also be constructed by passing a host pointer. The buffer is initialized with the data pointed to by a host pointer ("h"). The ownership of the memory is given to the buffer for the duration of its lifetime. An OpenCL memory object is released explicitly with the OpenCL function "clReleaseMemObject()". In contrast, the SYCL runtime will deallocate any storage required for the buffer when it is no longer in use. This may improve programming productivity since programmers are relieved of releasing memory objects manually in a complex program. However, understanding the implications of buffer destruction is required. Before the buffer is destroyed, the runtime will wait until all work on the buffer have completed, and then copy, if needed, the buffer content back to the host memory. The failure of constructing a SYCL buffer is reported as runtime exception.

TABLE II. MEMORY MANAGEMENT IN OPENCL AND SYCL

OpenCL	SYCL
d = clCreateBuffer(ctx, flags, BS, NULL, err)	buffer<T, D> d (WS)
d = clCreateBuffer(ctx, flags, BS, h, err)	buffer<T, D> d (h, WS)
clReleaseMemObject(d)	Handled by the SYCL runtime

TABLE III. DATA MOVEMENT BETWEEN HOST (SRC) AND DEVICE (DST) IN OPENCL AND SYCL

OpenCL	SYCL

// read from a buffer object to host
 	
clEnqueueReadBuffer(q, src,
  blocking_read, offset, cb, dst,
  0, 0, 0)
	q.submit([&] (handler &cgh) {
	  auto d = dst.get_access<sycl_read>(
	        cgh, range, offset);
	  cgh.copy(d, src);
	}).wait();

// write to a buffer object from host
 	
clEnqueueWriteBuffer(q, dst,
  blocking_write, offset, cb, src,
  0, 0, 0)
	q.submit([&] (handler &cgh) {
	  auto d = dst.get_access<sycl_write>(
	        cgh, range, offset);
	  cgh.copy(src, d);
	}).wait();

B. Data movement between a host and a device

Table III contrasts a migration path from OpenCL to SYCL for data transfers between a host and a device. "clEnqueueReadBuffer()" and "clEnqueueWriteBuffer()", enqueue commands to read from a buffer object to host memory and write to a buffer object from host memory in OpenCL, respectively. Both commands accept an offset in bytes (offset) and a data size in bytes (cb) being read from or written to. In contrast, a SYCL ranged accessor is constructed with a range starting at an offset from the beginning of the buffer. The "copy" method of the SYCL command-group handler (cgh) moves data between a device buffer and a host array through a buffer accessor. "sycl_read" and "sycl_write" are short names for the SYCL read and write access modes defined in the specification, respectively. The OpenCL read and write commands accept a parameter for blocking (synchronous) or non-blocking (asynchronous) data movement. In SYCL, the "wait()" method is called to wait for the asynchronous operation associated with the copy command to complete.

C. Coordinate indexing in ND-Range kernel

A kernel is typically offloaded to an accelerator to exploit its capability in parallel computing. A SYCL kernel is executed in a single-program-multiple-data manner where all work-items execute the same kernel program or instance in a *N*-dimensional range (ND-Range) [1]. Each work-item can query its location in a group that contains it and invoke functionalities specific to each group. The SYCL ND-Range covers the total execution range, which is divided into work-groups whose size must divide the ND-Range size in each dimension [5]. The SYCL "nd_item" class encapsulates information related to a work-item and a work-group [7]. Additionally, it contains barrier functions that act as synchronization points and must be encountered by all work-items in a work-group [1].

Table IV contrasts the coordinate index functions in a one-dimensional space (N = 1) and the barrier synchronization of memory operations to shared local memory [1] in the OpenCL and SYCL programs. "item" is an instance of the SYCL "nd_item" class. The names of the member functions are slightly different from those of the OpenCL functions.

TABLE IV. COORDINATE INDEX AND BARRIER IN OPENCL AND SYCL

OpenCL	SYCL
get_global_id(0)	item.get_global_id(0)
get_group_id(0)	item.get_group(0)
get_local_size(0)	item.get_local_range(0)
barrier(CLK_LOCAL_MEM_FENCE)	item.barrier(access::fence_space::local_space)

TABLE V. ATOMIC INCREMENT FUNCTION IN OPENCL AND SYCL

OpenCL	#pragma OPENCL EXTENSION cl_khr_global_int32_base_atomics : enable old_val = atomic_inc (var);

SYCL	template<typename T>
	T atomic_inc (T& val) {
	  atomic_ref<T, memory_order::relaxed, memory_scope::device,
	        access::address_space::global_space> obj (val);
	  return obj.fetch_add ((T)1);
	}

D. Atomic operation

Atomic operations allow for concurrent memory accesses from work-items in work-groups to a memory location without introducing data race. Multiple updates to a memory location do not overlap, but the order of updates is not deterministic.

Table V lists the OpenCL and SYCL atomic increment operation invoked in the compute kernels of the application. The OpenCL atomic function makes atomic increment on a variable in device global memory. The SYCL atomic reference class is instantiated with the type of the variable that it references, the memory order and scope, and the address space of the referenced object. The object is a reference to the value of the variable. While the expression of the SYCL atomic function is more verbose than the OpenCL function, the SYCL class builds on the OpenCL atomic access property and annotation [12] and extends it with a variety of atomic operations of both integer and floating-point types [7].

E. Kernel execution

An OpenCL kernel is defined using the "__kernel" declaration specifier. An address space qualifier may be used in variable declarations to specify the region of memory that is used to allocate the object [1]. All arguments to a kernel function shall be in the "__private" address space by default.

TABLE VI. EXECUTING THE FINDER KERNEL IN OPENCL AND SYCL

OpenCL	kernel void finder (__global char* chr,
	                     __constant char* pat,
	                      …
	                     __local char* l_pat,
	                     __local int* l_pat_index)
	{ // kernel body }
	
	clSetKernelArg(k, 0, …); // first kernel argument
	clSetKernelArg(k, 1, …); // second kernel argument
	…
	size_t gws[] = …; // global work size
	size_t lws[] = …; // local work size (work-group size)
	clEnqueueNDRangeKernel (q, k, 1, NULL, gws, lws, …);

SYCL	void finder (nd_item<1> &item,
	                char* chr, char* pat, …
	                char* l_pat, int* l_pat_index)
	{ // kernel body }
	
	range<1> gws (…);
	range<1> lws (…);
	q.submit([&](handler &h) {
	  h.parallel_for(nd_range<1>(gws, lws) [=] (nd_item<1> it) {
	    finder (it, …); // call the kernel function
	  });
	});

Function arguments declared to be a pointer of a data type can point to one of the following address spaces only: "__global", "__local" or "__constant". In contrast, the address spaces of the arguments of a SYCL kernel function declared to be pointers are inferred from the access targets of the SYCL accessors.

Before an OpenCL kernel is executed, the kernel's arguments need to be set with "clSetKernelArg()" properly as shown in Table VI. Then, "clEnqueueNDRangeKernel()" enqueues an OpenCL kernel to be executed on a device by specifying a command queue (q), a kernel object (k), dimension of an ND-Range kernel (1), global work size (gws), local work size (lws), and dependent events. Kernel launch is asynchronous, so it will return immediately after the kernel is enqueued in the command queue and likely before the kernel has even started execution. "clWaitForEvents()" or "clFinish()" is invoked to block execution on a host until the kernel completes. In SYCL, the global and local work sizes are specified using the SYCL range class [7]. 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 asynchronously a kernel to be executed on a device associated with the queue object. The "wait()" function waits for the event of the asynchronous operation to complete.

Since the methods of executing the two OpenCL kernels are similar, we will explain the migration process using the search kernel ("finder") as an example. The kernel argument "pat" in the OpenCL kernel is specified with a constant memory address space. In the SYCL program, a SYCL buffer is constructed whose content can be accessed through an accessor specialized with the "constant_buffer" access target. For the local memory arrays "l_pat" and "l_pat_index" accessed in the OpenCL kernel, we define two SYCL accessors with the corresponding types, dimensions, read and write access modes, and access target before the kernel is submitted. "sycl_read_write" and "sycl_lmem" are short names for the SYCL access mode and target [7], respectively. They indicate where and how data is accessed. Calling the function "finder" inside a lambda function in SYCL is not required, but the approach attempts to minimize code changes from OpenCL to SYCL.

IV. Experiments

A. Setup

We evaluate the performance of the OpenCL and SYCL applications on three computing systems with recent AMD GPUs. Major specifications of the Radeon VII (RVII), MI60, and MI100 discrete GPUs are listed in Table VII. The SYCL application currently executes on a single GPU device. The local work size (work-group size) is 256 for launching both SYCL kernels, whereas the sizes in the OpenCL program are determined by an OpenCL runtime. We build and execute the OpenCL application with the OpenCL support in the ROCm 4.5.2 [13]. The SYCL compiler is built from the SYCL branch of the Intel LLVM repository (04-08-2022) [14], and the version of the compiler frontend (Clang) is 15.0.0. The compiler optimization option is "-O3" for both applications. The host compilers are the GNU C compilers, versions 9.2 and above).

The datasets for our evaluation are the most recent assemblies of human genome, commonly nicknamed "hg38" and "hg19", from the UCSC genome sequences library [15]. "hg38" corrects thousands of small sequencing artifacts that cause false genetic variations, insertions, and deletions to be called when using "hg19" [16]. The input file, which contains the desired pattern, query sequences, and maximum mismatch number, is the same as the example listed in [17]. We run each executable four times and report the minimum elapsed time in seconds. The elapsed time excludes the setup of OpenCL and SYCL environments, reading the input file from a file system, or writing the headers to the output file.

B. Evaluation and Optimization

Table VIII lists the elapsed time in seconds of the OpenCL and SYCL applications on the GPUs for the two datasets. Comparing the execution time of the two applications shows that the performance speedup of the SYCL application over the OpenCL application across the GPUs ranges from 1 to 1.19.

While it is promising that performance portability of the SYCL application could be achieved on the target devices, we find that the "compare" kernel is a hotspot that accounts for approximately 98% of the total kernel execution time and 50% to 80% of the elapsed time on the GPUs. Hence, we will explore the kernel optimization for performance improvement.

Listing 1 shows the hotspot kernel that counts the number of mismatched bases in parallel. From line 0 (L0) to L8, the first thread in each work-group fetches the pattern (comp) and its index (comp_index) arrays sequentially from device global memory to shared local memory for data reuse. The lengths of both arrays are "plen × 2", which can accommodate two patterns from which one is selected based on the value of a flag. When the flag's value is 0 or 1 (L9), a local mismatch counter is reset to zero (L10). Then, each character in the first pattern is read at the indirect address "l_comp_index[j]" and compared against a set of values in the reference character at the address "loci[i] + l_comp_index[j]". When a mismatch occurs (L14), the counter is incremented by one (L15). The pattern comparisons will finish early when a mismatch threshold is reached (L16). When the mismatch count is not greater than the threshold (L19), the mismatch statistics, including the count, direction, and location, are stored at appropriate locations in device global memory. These locations can be computed in parallel using an atomic increment operation (L20 – L23). When the value of the flag is 0 or 2 (L26), each character of the second pattern is read at the address with an offset of "plen" and compared against a set of values of the reference character (L31). Since the comparison logics (L32 – L42) are almost the same as those for the first pattern, we will omit the explanation.

TABLE VII. Major Specifications of the GPUs (BW: bandwidth)

Device	*Global memory (GB)*	*GPU clock (MHz)*	*Memory clock (MHz)*	*Cores*	*L2 Cache (MB)*	*Peak BW (GB/s)*
RVII	16	1800	1000	3840	8	1024
MI60	32	1800	1000	4096	8	1024
MI100	32	1502	1200	7680	8	1228

TABLE VIII. Elapsed time of the OpenCL and SYCL Applications

Elapsed time (s)	**hg19**			**hg38**		
Device	*OCL*	*SYCL*	*speedup*	*OCL*	*SYCL*	*speedup*
RVII	54	48	1.12	71	61	1.16
MI60	51	50	1.02	63	63	1.00
MI100	49	41	1.19	61	58	1.05

```
void comparer(
  nd_item<1> &item,
  const unsigned int locicnts,
  const char* chr,
  const unsigned int* loci,
  unsigned int* mm_loci,
  const char* comp,
  const int* comp_index,
  unsigned int patternlen,
  unsigned short threshold,
  const char* flag,
  unsigned short* mm_count,
  char* direction,
  unsigned int* entrycount,
  char* l_comp,
  int* l_comp_index) {
0 int i = item.get_global_id(0);
1 unsigned int li = i - item.get_group(0) * item.get_local_range(0);
2 if (li == 0) {
3   for (k = 0; k < plen*2; k++) { // plen is pattern length
4     l_comp[k] = comp[k];
5     l_comp_index[k] = comp_index[k];
6   }
7 }
8 item.barrier(access::fence_space::local_space);
9 if (flag[i] == 0 || flag[i] == 1) {
10  lmm_count = 0;
11  for (j=0; j<plen; j++) {
12    k = l_comp_index[j];
13    if (k == -1) break;
14    if ((l_comp[k] == 'R' && (chr[loci[i]+k] == 'C' || chr[loci[i]+k] == 'T')) ||
          (l_comp[k] == 'Y' && (chr[loci[i]+k] == 'A' || chr[loci[i]+k] == 'G')) ||
          (l_comp[k] == 'K' && (chr[loci[i]+k] == 'A' || chr[loci[i]+k] == 'C')) ||
          (l_comp[k] == 'M' && (chr[loci[i]+k] == 'G' || chr[loci[i]+k] == 'T')) ||
          (l_comp[k] == 'W' && (chr[loci[i]+k] == 'C' || chr[loci[i]+k] == 'G')) ||
          (l_comp[k] == 'S' && (chr[loci[i]+k] == 'A' || chr[loci[i]+k] == 'T')) ||
          (l_comp[k] == 'H' && (chr[loci[i]+k] == 'G')) ||
          (l_comp[k] == 'B' && (chr[loci[i]+k] == 'A')) ||
          (l_comp[k] == 'V' && (chr[loci[i]+k] == 'T')) ||
          (l_comp[k] == 'D' && (chr[loci[i]+k] == 'C')) ||
          (l_comp[k] == 'A' && (chr[loci[i]+k] != 'A')) ||
          (l_comp[k] == 'G' && (chr[loci[i]+k] != 'G')) ||
          (l_comp[k] == 'C' && (chr[loci[i]+k] != 'C')) ||
          (l_comp[k] == 'T' && (chr[loci[i]+k] != 'T'))) {
15      lmm_count++;
16      if (lmm_count > threshold) break;
17    } // L14
18  } // L11
19  if (lmm_count <= threshold) {
20    old = atomic_inc(entrycount[0]);
21    mm_count[old] = lmm_count;
22    direction[old] = '+';
23    mm_loci[old] = loci[i];
24  } // L19
25 } // L9
26 if (flag[i] == 0 || flag[i] == 2) {
27   lmm_count = 0;
28   for (j=0; j<plen; j++) {
29     k = l_comp_index[plen + j];
30     if (k == -1) break;
31     if ((l_comp[k+plen] == 'R' && (chr[loci[i]+k] == 'C' || chr[loci[i]+k] == 'T')) ||
           … … (l_comp[k+plen] == 'T' && (chr[loci[i]+k] != 'T'))) {
32       lmm_count++;
33       if (lmm_count > threshold) break;
34     } // L31
35   } // L28
36   if (lmm_count <= threshold) {
37     old = atomic_inc(entrycount[0]);
38     mm_count[old] = lmm_count;
39     direction[old] = '-';
40     mm_loci[old] = loci[i];
41   } // L36
42 } // L26
```

Listing 1. Source of the "comparer" kernel in SYCL. The kernel counts the number of mismatched bases in parallel. It accounts for approximately 98% of the total kernel execution time. The entire Boolean conditions on Line 31 are similar to those on Line 14.

We explore the optimizations (opt1 – opt4) of the baseline hotspot kernel as follows. (1) We insert the "__restrict" keyword [18] in each pointer argument of the kernel function to prevent the compiler from creating unnecessary memory dependencies between non-conflicting memory load and store operations. (2) For each work-item, the base index of the reference character (loci[i]) and the value of the flag (flag[i]) are read from device global memory and stored in GPU registers first before they are used repeatedly for pattern comparisons. This may reduce the costly global memory accesses when a compiler fails to optimize the repeated memory accesses. (3) It is more efficient to fetch the pattern and its index arrays from device global memory to shared local memory when more work-items in a work-group participate in data fetching. (4) Fetching a pattern character from shared local memory (l_comp[k]) to a GPU register before it is accessed repeatedly for mismatch comparison may reduce the number of accesses to a shared local memory.

Figure 2 shows the kernel execution time in seconds with respect to the cumulative changes described in the last paragraph for the two datasets on the GPUs. Compared to the performance of the baseline kernel (base), removing pointer aliasing, registering the data read from global memory, and parallel data

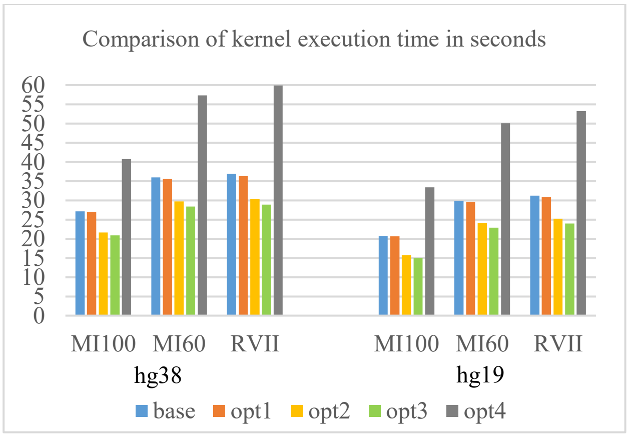

Fig. 2. Kernel execution time with respect to the proposed optimizations (opt1- opt4) for the two datasets on the AMD GPUs.

fetching from global memory to shared local memory are effective in performance improvement. For the "hg38" dataset, they reduce the time of the baseline kernel by 22.9%, 21.1%, and 21.7% on the three GPUs, respectively. For the "hg19" dataset, they reduce the kernel time by 27.8%, 23.4%, and 23.1% on the three GPUs, respectively. On the other hand, Table IX shows that the performance speedup from the kernel optimizations (opt3) ranges from 1.09 to 1.23 on the GPUs.

We attempt to better understand the performance implications of our optimizations through the resource usage of these kernels at the level of instruction-set architecture [19]. Table X lists the total instruction length in bytes of each kernel after it is compiled into assembly instructions, the number of scalar (S) and vector (V) general-purpose registers (GPRs) utilized by each kernel, and occupancy for each kernel. Occupancy is a measure of parallel work that a GPU could perform at a given time on a compute unit. Removing pointer aliasing reduces the code length by approximately 3.5%. Registering the global memory reads further reduces the code length by approximately 7.6%. Parallel data fetching from device global memory to shared local memory further reduces the code length by approximately 18.5%. In the meantime, the number of vector GPRs decrease from 64 to 57 and the number of scalar GPRs from 22 to 10. While registering the shared local memory read can further reduce the code length by approximately 17%, it increases the usage of scalar registers from 57 to 82. While the pressure of register usage causes occupancy to decrease only from 10 to 9, the kernel execution time almost doubles on the GPUs as shown in Figure 2. The results show that occupancy has a significant impact on the

TABLE IX. ELAPSED TIME OF THE OPTIMIZED SYCL APPLICATION

Elapsed time (s)	**hg19**			**hg38**		
Device	*base*	*opt*	*speedup*	*base*	*opt*	*speedup*
RVII	48	39	1.23	61	52	1.17
MI60	50	42	1.19	63	57	1.11
MI100	41	36	1.14	58	53	1.09

TABLE X. RESOURCE USAGE AND OCCUPANCY OF THE KERNELS

Metrics	**base**	**opt1**	**opt2**	**opt3**	**opt4**
Code length	6064	5852	5408	4408	3660
#SGPRs	64	64	64	57	82
#VGPRs	22	22	22	10	10
Occupancy	10	10	10	10	9

performance of the kernel; there is a performance trade-off between register usage and occupancy on the GPUs.

V. Related Work

In [9], the authors introduced the OpenCL implementation of Cas-OFFinder and evaluated its performance on an Intel i7 3770K CPU and an AMD Radeon HD7878 GPU. The OpenCL program sees significant speedup of the searching process using the GPU. FlashFry [20], a tool written in Scala for characterizing large numbers of target sequences, ran approximately two to three orders of magnitude faster than Cas-OFFinder on an Intel Xeon CPU. The authors of Cas-OFFinder optimized the OpenCL kernels with a 2-bit sequence format, shared local memory and atomic operations, and parallel computing with OpenMP in the host program. These optimizations can reduce memory access latency and increase data parallelism, improving the performance of the application by a factor of 30 approximately [21]. The current OpenCL and SYCL kernels include these optimizations. In [22], the authors proposed an architecture-specific method for finding potential guided RNA off-target. The theoretical speedup reported by the method was significant compared to other CPU and GPU implementations. Their study was focused on a specialized hardware architecture that is a good fit for the application. We focus on the development and improvement of the application with heterogeneous programming models on general-purpose accelerators. In [23], the authors proposed a fast off-target detection tool that could improve query speed and reduce memory usage with the method of FM index in off-target searching. They evaluated the implementation on an Intel Xeon CPU. Further work will be needed to understand the performance of the proposed method on a GPU.

VI. Conclusion

SYCL is a single-source C++ programming layer that extends the concepts, portability, and efficiency of OpenCL. In this paper, we choose a popular off-target detection application in bioinformatics for studying performance and portability of SYCL. We explain the experience of migrating the application from OpenCL to SYCL and evaluate the performance of the SYCL applications on AMD GPUs. The experimental results show that the SYCL program can achieve comparable or higher performance compared to the OpenCL program on the GPUs. Exploring the optimizations of SYCL kernels show that eliminating pointer aliasing, registering repeatedly data accesses from device global memory, and fetching data from global device memory to shared local memory by all work-items in a work-group are effective in performance improvement. We hope that our results and experiences are valuable to migrating applications from OpenCL to SYCL.

Acknowledgment

We appreciate the anonymous reviewers for their comments and suggestions, and Ada Sedova from Oak Ridge National Laboratory (ORNL) for the direction to the application. The research used resources at the Experimental Computing Lab at ORNL. This research was supported by the US Department of Energy Advanced Scientific Computing Research program under Contract No. DE-AC05-00OR22725.

References

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

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

[3] 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.

[4] 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.

[5] 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.

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

[7] SYCL 2020 Specification (revision 4). [https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html.

[8] Zhang, F., Wen, Y. and Guo, X., 2014. CRISPR/Cas9 for genome editing: progress, implications and challenges. Human molecular genetics, 23(R1), pp.R40-R46.

[9] Bae, S., Park, J. and Kim, J.S., 2014. Cas-OFFinder: a fast and versatile algorithm that searches for potential off-target sites of Cas9 RNA-guided endonucleases. Bioinformatics, 30(10), pp.1473-1475.

[10] Shah, S.A., Erdmann, S., Mojica, F.J. and Garrett, R.A., 2013. Protospacer recognition motifs: mixed identities and functional diversity. RNA biology, 10(5), pp.891-899.

[11] Liu, G., Zhang, Y. and Zhang, T., 2020. Computational approaches for effective CRISPR guide RNA design and evaluation. Computational and structural biotechnology journal, 18, pp.35-44.

[12] Sorensen, T., Donaldson, A.F., Batty, M., Gopalakrishnan, G. and Rakamarić, Z., 2016, October. Portable inter-workgroup barrier synchronisation for GPUs. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (pp. 39-58).

[13] ROCm Open Ecosystem, 2021. AMD. [online] Available: https://www.amd.com/en/graphics/servers-solutions-rocm

[14] Intel staging area for llvm.org. [online] Available: https://github.com/intel/llvm

[15] UCSC genome sequences library. [online] Available: http://hgdownload.soe.ucsc.edu/downloads.html.

[16] Human-genome-reference-builds, GATK technical documentation Glossary, 2022

[17] An ultra fast and versatile algorithm that searches for potential off-target sites of CRISPR/Cas-derived RNA-guided endonucleases., 2021. [online] Available: https://github.com/snugel/cas-offinder

[18] Wen-mei, W.H., 2015. Heterogeneous System Architecture: A new compute platform infrastructure. Morgan Kaufmann.

[19] AMD Instinct MI100 Instruction Set Architecture [online] Available:https://developer.amd.com/wpcontent/resources/CDNA1_Shader_ISA_14December2020.pdf

[20] McKenna, A., Shendure, J. FlashFry: a fast and flexible tool for large-scale CRISPR target design. BMC Biol 16, 74 (2018

[21] Park, J., Bae, S. and Kim, J.S., 2015. Cas-Designer: a web-based tool for choice of CRISPR-Cas9 target sites. Bioinformatics, 31(24), pp.4014-4016.

[22] Bo, C., Dang, V., Sadredini, E. and Skadron, K., 2018, February. Searching for potential gRNA off-target sites for CRISPR/Cas9 using automata processing across different platforms. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA) (pp. 737-748). IEEE.

[23] Cui, Y., Liao, X., Peng, S., Tang, T., Huang, C. and Yang, C., 2020. OffScan: a universal and fast CRISPR off-target sites detection tool. BMC genomics, 21(1), pp.1-6.