

Understanding Performance Portability of Bioinformatics Applications in SYCL on an NVIDIA GPU

Zheming Jin
Oak Ridge National Laboratory
jinz@ornl.gov

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

Abstract— Our goal is to have a better understanding of performance portability of SYCL kernels on a GPU. Toward this goal, we migrate representative kernels in bioinformatics applications from CUDA to SYCL, evaluate their performance on an NVIDIA GPU, and explain the performance gaps through performance profiling and analyses. We hope that the findings provide valuable feedback to the development of the SYCL ecosystem.

Keywords—Bioinformatics, Performance portability, Compiler optimization, Heterogeneous computing

I. INTRODUCTION

Since graphics-processing unit (GPU) computing platforms are different in the details of vendors' hardware architectures and software stacks [1, 2, 3], vendor-specific programming libraries and languages have been addressing the differences. For example, CUDA [4] is an advanced programming model widely used for NVIDIA GPUs. However, commonalities among these programming models exist and several portable programming approaches allow for writing code that supports multiple target platforms [5]. In this study, we focus on SYCL, an evolving programming model that facilitates portability across vendors' devices [6].

To evaluate function and performance portability across these devices, previous studies compare the performance of benchmarks and applications having both SYCL and CUDA implementations on NVIDIA GPUs. The results indicate that whether the performance of running SYCL is competitive with using CUDA directly depends on the applications and how they are optimized [7, 8, 9, 10]. For example, migrating highly optimized bioinformatics kernels (compute-intensive routines) in CUDA requires significant code changes, and the SYCL implementation is 2X slower [8]. Therefore, functional and performance portability was not achieved fully. After migrating the Rodinia benchmark suite [11] from CUDA to SYCL, the researchers find that some SYCL kernels achieve performance portability and other kernels see considerable overhead, varying from 25% to 190%, due to their execution of more GPU instructions and/or underutilization of GPU resources [10]. In [12], the authors describe their experiences of porting NAMD, a large molecular dynamics software application. While porting most CUDA kernels in the application is straightforward, the library-based approach for migrating the primitive operations, such as reduction, scan, and sort, implies that the performance of the SYCL libraries may be critical to the overall performance of the applications. Some researchers also recommend CUDA when the target hardware is an NVIDIA GPU only [13]. The reason is that CUDA's mature development environment and its variety of libraries can speed up the development process. It is also easy to find detailed examples and help online. These

studies indicate that there is a need of improving performance portability by understanding performance gaps at first.

A language and compiler may only guarantee that they can make it a little easier for developers to achieve portability [14]. Hence, achieving portability is nontrivial for a portable language such as SYCL. Toward the goal of improving performance portability, our work is to identify performance gaps in bioinformatics applications that could be improved with the development of the SYCL ecosystem. We attempt to obtain a better understanding of the causes of performance gaps by analyzing representative kernels in bioinformatics applications. More specifically, we select open-source applications with public code repositories, migrate them from CUDA to SYCL, evaluate the performance of these programs on an NVIDIA GPU, identify the performance gaps with profiling and analyses, and provide suggestions for improving performance portability.

We have a few suggestions based on the findings of our study. The SYCL compiler may optimize address generation for shared local memory (SLM) using 32-bit addresses and simple arithmetic operations. The compiler may prove that computing an address offset using 32-bit multiplication will not cause an overflow of the result. The compiler may select a vector load instruction for eliminating byte extraction that would incur an overhead of eight instructions. The compiler may unroll a loop more automatically without a user's manual direction to improve the performance of a kernel. The performance of a SYCL library interface may be evaluated carefully to minimize the overhead caused by a feature implementation that is hidden from a user of the library. In addition, the choices of the ordering of a SYCL queue object and the SYCL math function are important for performance portability.

While the experimental results are based on a limited number of bioinformatics programs, the selected kernels may be representative of GPU kernels using optimized memory footprints and global memory accesses, shared local memory, loop(s) for iterating over attributes and elements, fast math library, or vendor-specific libraries for productivity and performance. We have described the motivation and scope of our work. The remainder of the paper is organized as follows. The next section introduces the SYCL programming model and the kernels in the bioinformatics applications briefly. Section III describes the results of evaluating and analyzing these kernels on a GPU. Section IV concludes the paper.

II. BACKGROUND

A. Introduction to SYCL

Open Computing Language (OpenCL) is a standard maintained by the Khronos group. It has facilitated the

development of parallel computing programs for execution on CPUs, GPUs, other accelerators [15,16]. While OpenCL is a mature programming model, an OpenCL program is much more verbose than a CUDA program. Hence, writing an OpenCL program tends to be tedious and error-prone [17,18]. Built on the underlying concepts, portability, and efficiency of OpenCL and ease of use and flexibility of single-source C++ [19], SYCL is a programming language for heterogeneous computing across devices of multiple vendors. Combining a host program and a device program allows a programmer to gain the simplicity of writing a single program and a compiler to statically type-check the correctness of the program. SYCL buffers and unified shared memory (USM) are two abstractions for data management [10]. The SYCL USM allows for more straightforward migration of existing CUDA programs while the SYCL buffers relieve the burden of explicit memory management from a researcher. The SYCL compiler with CUDA support is part of a large open-source project [20] for the implementations of the evolving SYCL standard [6]. Because the capabilities of the OpenCL implementation from NVIDIA are limited, the plugin interface was adopted and it does not rely on the OpenCL support from NVIDIA, facilitating more features and potentially higher overall performance [21].

B. Introduction to the bioinformatics kernels

In this study, we choose open-source bioinformatics programs accelerated with CUDA on a GPU. In general, we focus on GPU kernel execution in these programs. The first kernel (k1) implements all-pairs distance where a distance is computed between a pair of instances. The distance may be the number of attributes that differ between two instances in single nucleotide polymorphism (SNP) data. In this study, we select the kernel optimized with coalesced memory accesses, reduced memory footprints and global memory accesses, and memory hierarchy composed of registers and SLM [22].

The second kernel (k2) evaluates SNP combinations for explaining complex traits in epistasis detection [23]. It consists of counting the frequencies of genotypes for each unique set of SNPs, scoring each set of SNPs based on the frequencies table, and selecting the local optimal solution with the minimum score from the SNP indexes [24].

The third kernel (k3) computes a more accurate p -value by generating many pseudorandom numbers in gene-expression connectivity mapping [25]. Generating many pseudorandom signatures on a host was identified as a performance bottleneck of the application [26]. The GPU application takes advantage of a CUDA-specific pseudorandom number generation library (cuRAND) to reduce the execution time significantly [27].

The fourth kernel (k4) is a parallel implementation of the Needleman-Wunsch algorithm targeting a GPU. The algorithm finds the global optimal alignment of two biosequences after structuring the pair as a two-dimensional matrix. The optimized kernel takes advantage of thread-block level of parallelism and program locality using SLM on a GPU to reduce global memory accesses and overhead of kernel execution [11].

The last kernel (k5) calculates the Minkowski distance, a distance metric used in bioinformatics applications such as compound classification [28], identification of significant gene

sets associated with a disease [29], and k -means clustering frameworks [30]. The CUDA and SYCL kernels, which compute the distances of points stored as two matrices, are available in [31].

III. EVALUATION AND ANALYSES

Previous studies have described in detail the experience of migrating CUDA programs to SYCL in memory management, intrinsic and arithmetic functions, atomic functions, and kernel execution [9, 32]. Hence, we will skip the lengthy descriptions of migrating these bioinformatics applications from CUDA to SYCL and focus on understanding performance gaps in performance portability. The SYCL programs are compiled using the SYCL compiler with CUDA support from the Intel LLVM release 2022-06. The CUDA programs are compiled using the NVIDIA HPC SDK 22.7. The optimization option of the compilers is “O3”. The execution time of the kernels are measured on an NVIDIA Tesla V100 DGXS GPU. The operation system is Ubuntu 20.04. For k1, the numbers of instances and attributes are 224 and 4096, respectively. For k2, there are 4096 SNPs and 8192 samples. For k3, there are 1,000,000 pseudorandom signature generations. For k4, the length of both sequences is 16,384. For k5, there are 1,048,576 points and each point has a dimension of 1024.

A. The all-pairs distance calculation kernel (k1)

Listing 1 shows the distance calculation kernel in SYCL. The word size of the SLM is THREADS, which is a constant value of 128. The size is equal to the work-group size. From line 4 (L4) to L6, the kernel queries work-item and work-group identifiers (IDs). L7 initializes the SLM’s content and L8 waits for the initialization to complete for all work-items in a work-group. Inside the loop body (L10 to L17), attributes in bytes are loaded as four-element vectors from a device global memory.

```

1 q.submit([&] (handler &h) {
2   accessor<int,1,sycl_read_write,
   access::target::local> dist(THREADS, h);
3   h.parallel_for(nd_range<2>(gws,lws),
   [=] (nd_item<2> item) {
4     int idx = item.get_local_id(1);
5     int gx = item.get_group(1);
6     int gy = item.get_group(0);
7     dist[idx] = 0;
8     item.barrier(fence_space::local_space);
9     for(int i = idx*4;
   i < ATTRIBUTES; i+=THREADS*4) {
10      char4 j = *(char4 *) (
   d_data_char + i + ATTRIBUTES*gx);
11      char4 k = *(char4 *) (
   d_data_char + i + ATTRIBUTES*gy);
12      int count = 0;
13      if(j.x() ^ k.x()) count++;
14      if(j.y() ^ k.y()) count++;
15      if(j.z() ^ k.z()) count++;
16      if(j.w() ^ k.w()) count++;
17      dist[idx] += count;
18    }
19  });
20 }.wait();

```

Listing 1. All-pairs distance calculation kernel in SYCL. Only the relevant codes are shown here for clarity. The values of ATTRIBUTES and THREADS are fixed.

Then, they are compared for equality in values. Finally, the number of unequal attributes is counted to compute the distance, and the distance is saved in SLM for each work-item. We omit the remaining parts of the kernel as they are not of interest in our discussion.

Evaluating the performance of the two kernels indicates that the CUDA kernel is 25% faster than the SYCL kernel on the GPU. For a better understanding of the performance gap, we instruct the CUDA and SYCL compilers to generate the parallel thread execution (PTX) codes for both kernels. PTX defines a virtual machine and instruction-set architecture for general-purpose parallel thread execution [33]. Then, we analyze the codes to understand the differences in code generation and compiler optimization.

Listings 2 and 3 show the PTX codes for the initialization of the SLM in the CUDA and SYCL kernels, respectively. `%tid.x` contains the thread ID value (L1). Calculating the byte address of the SLM is implemented as a 32-bit shift left operation (L2) by the CUDA compiler whereas it is a wide multiply instruction (L2) by the SYCL compiler. In general, the simple shift operation is faster than the complex multiply

```

1  mov.u32      %r3, %tid.x;
2  shl.b32     %r94, %r3, 2;
3  mov.u32     %r99, slm_dist;
4  add.s32    %r4, %r99, %r94;
5  mov.u32    %r35, 0;
6  st.shared.u32 [%r4], %r35;

```

Listing 2. The PTX codes generated by the CUDA compiler represent the operations $d_dist[idx] = 0$. The symbol `dist` represents the base address of the shared memory used in the kernel. The memory is addressed using 32-bit arithmetic operations.

```

1  mov.u32      %r1, %ctaid.x;
2  mov.u32      %r2, %ctaid.y;
3  mov.u32      %r3, %tid.x;
4  ld.param.u64 %rd18, [param];
5  cvta.to.global.u64 %rd1, %rd18;
6  shl.b32     %r37, %r1, 12;
7  cvt.s64.s32 %rd2, %r37;
8  shl.b32     %r38, %r2, 12;
9  cvt.s64.s32 %rd3, %r38;

```

Listing 4. The PTX codes generated by the CUDA compiler represent the operations $d_data_char + ATTRIBUTES * gx$ and $d_data_char + ATTRIBUTES * gy$. The constant value of `ATTRIBUTES` is 4096. The `m`-multiplications are implemented as 32-bit shift left operations without a post-multiply mask.

```

1  ld.global.v4.u8{%rs1,%rs2,%rs3,%rs4}, [%rd29];
2  ld.global.v4.u8{%rs5,%rs6,%rs7,%rs8}, [%rd30];
3  setp.ne.s16  %p3, %rs1, %rs5;
4  selp.u32    %r43, 1, 0, %p3;
5  setp.eq.s16  %p4, %rs2, %rs6;
6  selp.b32    %r44, 2, 1, %p3;
7  selp.b32    %r45, %r43, %r44, %p4;
8  setp.ne.s16  %p5, %rs3, %rs7;
9  selp.u32    %r46, 1, 0, %p5;
10 setp.ne.s16 %p6, %rs4, %rs8;
11 selp.u32    %r47, 1, 0, %p6;

```

Listing 6. The PTX codes generated by the CUDA compiler represent the vector load, comparison, and count increments in Listing 1.

operation. The base address of the SLM is stored as a 32-bit number (L3) by the CUDA compiler and it is treated as a 64-bit number (L3) by the SYCL compiler. Consequently, byte addresses to the SLM for each work-item are computed with 32-bit additions in the CUDA kernel and 64-bit additions in the SYCL kernel. Comparing the PTX codes for addressing the SLM suggests that the SYCL compiler better optimize the generation of the SLM's addresses using 32-bit operations and simple arithmetic operations.

Listings 4 and 5 show the PTX codes for the generation of global memory addresses (L10 and L11 in Listing 1) in the CUDA and SYCL kernels, respectively. We will explain the PTX codes of the CUDA kernel first. The IDs of work-groups and work-items are stored as 32-bit numbers in three registers (L1-3). L4 fetches the base address of the device array as a 64-bit number and stores it in a register. L5 converts the address between `generic` and `global` state spaces. Two 32-bit shift left operations (L6 and L8) are performed on the work-group IDs to compute the results of $gx \times 4096$ and $gy \times 4096$, respectively. Finally, the results are converted to 64-bit numbers. In the SYCL kernel, the work-group IDs are converted to 64-bit values in the beginning on L2 and L4. Consequently,

```

1  mov.u32      %r1, %tid.x;
2  mul.wide.u32 %rd14, %r1, 4;
3  mov.u64     %rd15, slm_dist;
4  add.s64    %rd3, %rd15, %rd14;
5  mov.u32    %r12, 0;
6  st.shared.u32 [%rd3], %r12;

```

Listing 3. The PTX codes generated by the SYCL compiler represent the operations $d_dist[idx] = 0$. The symbol `dist` represents the base address of the shared memory used in the kernel. The memory is addressed using 64-bit arithmetic operations.

```

1  mov.u32      %r10, %ctaid.y;
2  cvt.u64.u32 %rd1, %r10;
3  mov.u32      %r11, %ctaid.x;
4  cvt.u64.u32 %rd2, %r11;
5  ld.param.u64 %rd12, [param];
6  shl.b64     %rd16, %rd2, 12;
7  and.b64     %rd17, %rd16, 4294963200;
8  shl.b64     %rd18, %rd1, 12;
9  and.b64     %rd19, %rd18, 4294963200;
10 add.s64    %rd5, %rd12, %rd19;
11 add.s64    %rd6, %rd12, %rd17;

```

Listing 5. The PTX codes generated by the SYCL compiler represent the operations $d_data_char + ATTRIBUTES * gx$ and $d_data_char + ATTRIBUTES * gy$. The constant value of `ATTRIBUTES` is 4096.

```

1  ld.global.u32 %r13, [%rd20];
2  cvt.u16.u32  %rs1, %r13;
3  and.b16     %rs2, %rs1, 255;
4  shr.u16     %rs3, %rs1, 8;
5  shr.u32    %r14, %r13, 16;
6  cvt.u16.u32 %rs4, %r14;
7  and.b16     %rs5, %rs4, 255;
8  shr.u32    %r15, %r13, 24;
9  cvt.u16.u32 %rs6, %r15;
10 add.s64    %rd21, %rd5, %rd29;
11 ld.global.u32 %r16, [%rd21];

```

Listing 7. The PTX codes generated by the SYCL compiler represent the vector load and the overhead of byte extraction.

the generation of the addresses requires a sequence of 64-bit arithmetic operations. After the 64-bit shift left operations, L7 and L9 obtain the lower 32 bits of the 64-bit values by a bitmask in decimal value. The hexadecimal value of the mask is 0xFFFF_F000. Comparing the results of address generation by the SYCL and CUDA compilers suggests for the SYCL compiler to prove that computing an address offset using 32-bit multiplication will not cause an overflow of the product.

Listing 6 shows the PTX codes corresponding to the vector load using a `char4` data type and equality counting in the CUDA kernel. Each vector load instruction (L1 and L2) reads four bytes as a vector from a device memory, and then stores them in four distinct registers. The conditions computed using exclusive disjunction operations in the kernel are mapped to the comparison and selection instructions. This eliminates branch instructions that may cause branch divergence among work-items. Retrieving four components of a vector in the kernel corresponds to four register reads in the instructions. After the execution of selection instructions, the registers `r43`, `r44`, `r45`, `r46`, `r47` contain the values of 0, 1, or 2. Adding them up (not shown in the listing) is equivalent to the conditional counting in the kernel.

Listing 7 shows the PTX codes for the vector load and equality counting in the SYCL kernel. The first load instruction reads a single 32-bit number from the memory and stores it in a register. Then, a sequence of conversion, logical, and shift instructions (L2 to L10) are executed to extract four distinct bytes from the 32-bit value. These operations are not necessary when the SYCL compiler can generate a vector load instruction. After the extraction is done, the second load instruction is executed (L11) in the same manner. When all vector elements are ready, the comparison and selection instructions, as shown in Listing 6, are executed. Hence, they are omitted here. Comparing the PTX codes generated by the compilers suggest for the SYCL compiler to select a vector load instruction for retrieving vector components from registers directly. This can eliminate byte extraction that requires eight instructions.

B. The epistasis detection kernel (k2)

Evaluating the CUDA and SYCL applications indicates that the execution time of the SYCL application is almost 3.5X longer than that of the CUDA application. After analyzing the PTX codes generated by the two compilers, we find that the performance bottleneck is caused by the nested loops shown in Listing 8. The CUDA compiler unrolls the two loops fully while

```

1 const int pow_table[10] =
  {1,3,9,27,81,243,729,2187,6561,19683};
2 unsigned long long casesArr[3 * EPISTASIS_SIZE];
3 for(int i = 0; i < COMB_SIZE; i++) {
4   unsigned long long acc = 0xFFFFFFFFFFFFFFFF;
5   for(int j=0; j < EPISTASIS_SIZE; j++) {
6     acc = acc & casesArr[j * 3 +
7       ((int) (i / pow_table[j])) % 3];
8   } // inner loop
9   observedValues_shared[i*2*WORKGROUP_SIZE +
  WORKGROUP_SIZE + local_id] += __popc11(acc);
10 } // outer loop

```

Listing 8. The nested loops in the CUDA epistasis kernel. The loops in the SYCL kernel are not fully unrolled by the SYCL compiler.

the SYCL compiler does not. The trip count of the outer loop is `COMB_SIZE`, a predefined constant of value 27. The trip count of the inner loop is `EPISTASIS_SIZE`. Its value is 3. To unroll the loops fully, the CUDA compiler converts the 10-element read-only `power` table and the 9-element `case` array to GPU registers, precomputes the arithmetic expression `j * 3 + (int)(i / pow_table[j]) % 3` for indexing the SLM, and selects memory index appropriately for each unrolled loop iteration. Loop unrolling reduces the dynamic instruction count due to the fewer number of compare and branch operations for the same amount of work. It also provides more instruction scheduling opportunities with the availability of additional independent instructions for increasing instruction-level parallelism [34].

The nested loops are not unrolled fully by the SYCL compiler using the same compiler optimization option. We also find that the loop (starting from L9) in the first kernel (k1) is not unrolled automatically by the compiler whereas the CUDA compiler unrolls it by a factor of four. The SYCL compiler has a threshold for loop unrolling. When the threshold is exceeded, a loop will not be unrolled automatically by the compiler. In this case, we could direct the SYCL compiler to unroll the nested loops fully to achieve performance portability in terms of loop optimization. The results suggest that the SYCL compiler automatically unroll a loop that may benefit from branch reduction and instruction-level parallelism.

C. The *p*-value compute kernel (k3)

We migrate the connectivity mapping application from CUDA to SYCL with the support of the math kernel library (oneMKL) interface for pseudorandom number generation [35] using a vendor-specific library. Performance profiling shows that computing the *p*-value is the most time-consuming step when iterating over the reference gene-expression profiles [25]. The SYCL kernel is about 2.1X slower than the CUDA kernel. While both libraries invoke the same routines in the CUDA pseudorandom library for generating sequenced numbers and seeds, the oneMKL interface adds an extra kernel called in the function `range_transform_fp()` [36]. Highlighted in

```

virtual inline void generate(
  const uniform<float,uniform_method::standard>&
  distr, std::int64_t n,
  sycl::buffer<float, 1>& r) override
{
  q.subm-it([&](sycl::handler& cgh) {
    auto acc = r.get_access<sycl_read_write>(cgh);
    cgh.host_task(=[&](sycl::interop_handle ih) {
      auto r_ptr =
        reinterpret_cast<float*>(ih.get_native_mem
        <sycl::backend::ext_oneapi_cuda>(acc));
      curandStatus_t status;
      CURAND_CALL(curandGenerateUniform,
        status, engine_, r_ptr, n);
    });
  }).wait_and_throw();
  range_transform_fp<float>(q, distr.a(),
    distr.b(), n, r);
}

```

Listing 9. Implementation of the oneMKL pseudorandom number generation function with the `cuRAND` function as the backend. The namespace `"oneapi::mkl::rng"` is omitted for clarity.

Listing 9, the function facilitates a custom range for sampling pseudorandom numbers with a uniform distribution. However, executing an extra kernel leads to a performance drop. Our analysis suggests that the performance of a SYCL library interface be evaluated carefully to minimize the overhead caused by a feature implementation that is hidden from a user of the library.

D. The Needleman-Wunsch kernel (k4)

Evaluating the total execution time of the kernels shown in Listing 10 on the GPU shows that the SYCL implementation is approximately 1.8X slower than the CUDA implementation. The slowdown is caused by the runtime overhead of kernel execution when the SYCL queue executes these kernels (tasks) in any order (i.e., out-of-order) subject to data dependencies among these tasks. On the other hand, an in-order queue executes tasks in the order in which they are submitted. We evaluate the impact of the execution orders by profiling the SYCL program running on the GPU for a better understanding of the overhead.

Table I lists the names of the CUDA driver functions [37] of interest and their call counts with respect to the two execution orders. Comparing the number of function calls shows that an out-of-order queue incurs significant costs of waiting for events to complete and stream synchronization at runtime. An out-of-order is preferred when no dependence exists among independent tasks. However, there is a tradeoff between task parallelism and runtime overhead in the implementation of the SYCL runtime. The impact of the runtime overhead upon the

```

for (int blk = 1; blk <= block_width; blk++) {
    gws = BLOCK_SIZE * blk;
    q.submit([&](handler& cgh) {
        ...
        cgh.parallel_for<class kernel1>(
            nd_range<1>(range<1>(gws), range<1>(lws)),
            [=] (nd_item<1> item) {
                // first kernel
            });
    });
}
for (int blk = block_width-1; blk >= 1; blk--) {
    gws = BLOCK_SIZE * blk;
    q.submit([&](handler& cgh) {
        ...
        cgh.parallel_for<class kernel2>(
            nd_range<1>(range<1>(gws), range<1>(lws)),
            [=] (nd_item<1> item) {
                // second kernel
            });
    });
}

```

Listing 10. Execution of the two SYCL kernels in the Needleman-Wunsch benchmark (block_width = 1024)

TABLE I. RUNTIME OVERHEAD FOR EXECUTING K4 WITH AN OUT-OF-ORDER QUEUE IN TERMS OF THE NUMBER OF FUNCTION CALLS

Runtime functions	In-order	Out-of-order
cuStreamCreate	1	131
cuStreamWaitEvent	0	4096
cuStreamSynchronize	3	453
cuStreamDestroy	1	131
cuCtxGetCurrent	8214	12309

```

1    lg2.approx.ftz.f32    %f18, %f17;
2    mul.ftz.f32         %f19, %f18, %f8;
3    ex2.approx.ftz.f32  %f20, %f19;

```

Listing 11. The PTX instructions generated for the single-precision floating-point power function $\text{powf}(x, y)$; the CUDA compiler converts the power function to $2^{y \times \log_2(x)}$ when the fast math library is enabled.

performance of a kernel or an application may be negligible when a kernel is executed once. When the kernel is executed many times, we could reduce the overhead of out-of-order execution using an in-order queue.

E. The Minkowski distance kernel (k5)

Evaluating the kernel performance on the GPU shows that the SYCL kernel is more than 5X slower than the CUDA kernel. When the fast math library is enabled, the CUDA compiler converts the single-precision floating-point power function $\text{powf}(x, y)$ to $2^{y \times \log_2(x)}$. Hence, only three PTX instructions, as shown in Listing 11, are generated for logarithm, multiplication, and exponentiation, respectively. It is straightforward to map the CUDA math function to the SYCL math function `sycl::pow(x, y)`. However, the SYCL compiler does not optimize the math function even when the fast math library is enabled. To direct the compiler to generate the optimal number of instructions for the math function, `sycl::powr(x, y)` should be used. According to the SYCL specification, x is a non-negative number expected by the function. Alternatively, the program can call the SYCL function `sycl::native::powr(x, y)` when the fast math library is not used. In a sense, the CUDA intrinsic `__powf(x, y)` is equivalent to the SYCL function defined in the `sycl::native` namespace. The performance comparison suggests that it is important to choose a math function properly in the SYCL math library for performance portability. An intimate knowledge of the math libraries and benchmarking the performance of math functions in SYCL and CUDA may help application developers make a right choice in the beginning.

IV. CONCLUSION

Achieving performance portability is a challenging task for a portable programming model. In this study, we attempt to have a better understanding of performance portability of SYCL kernels in bioinformatics applications. Through performance profiling and analyses, we identify and explain the performance gaps of CUDA and SYCL kernels when they execute on an NVIDIA GPU. We hope that the findings will be valuable to compiler, library, and application developers. In our future work, we will evaluate more CUDA and SYCL kernels in bioinformatics applications for understanding performance portability.

ACKNOWLEDGMENT

The author would like to thank the Codeplay developers for explaining the SYCL math function and the reviewers for their comments and suggestions. The research used resources at 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] Lindholm, E., Nickolls, J., Oberman, S. and Montrym, J., 2008. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2), pp.39-55.
- [2] Gutierrez, A., Beckmann, B.M., Dutt, A., Gross, J., LeBeane, M., Kalamatianos, J., Kayiran, O., Poremba, M., Potter, B., Puthoor, S. and Sinclair, M.D., 2018, February. Lost in abstraction: Pitfalls of analyzing GPUs at the intermediate language level. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA) (pp. 608-619). IEEE.
- [3] Blythe, D., 2020, August. The Xe GPU Architecture. In 2020 IEEE Hot Chips 32 Symposium (HCS) (pp. 1-27). IEEE Computer Society.
- [4] 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.
- [5] Portability Across DOE Office of Science HPC Facilities. [online] <https://performanceportability.org/>
- [6] SYCL 2020 Specification (revision 5) [online] <https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html>
- [7] 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).
- [8] Haseeb, M., Ding, N., Deslippe, J. and Awan, M., 2021, November. Evaluating Performance and Portability of a core bioinformatics kernel on multiple vendor GPUs. In 2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC) (pp. 68-78). IEEE
- [9] Jin, Z. and Vetter, J.S., 2022, August. Performance portability study of epistasis detection using SYCL on NVIDIA GPU. In Proceedings of the 13th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics (pp. 1-8).
- [10] 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.
- [11] 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.
- [12] Hardy, D.J., Choi, J., Jiang, W. and Tajkhorshid, E., 2022, May. Experiences Porting NAMD to the Data Parallel C++ Programming Model. In International Workshop on OpenCL (pp. 1-5).
- [13] Marcel Breyer, Alexander Van Craen, and Dirk Pflüger. 2022. A Comparison of SYCL, OpenCL, CUDA, and OpenMP for Massively Parallel Support Vector Machine Classification on Multi-Vendor Hardware. In International Workshop on OpenCL (IWOCL'22). Association for Computing Machinery, New York, NY, USA, Article 2, 1–12. <https://doi.org/10.1145/3529538.3529980>
- [14] 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.
- [15] Munshi, A., Gaster, B., Mattson, T.G. and Ginsburg, D., 2011. OpenCL programming guide. Pearson Education.
- [16] Kaeli, D., Mistry, P., Schaa, D. and Zhang, D.P., 2015. Heterogeneous computing with OpenCL 2.0. Morgan Kaufmann.
- [17] 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.
- [18] 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.
- [19] Stroustrup, B., 2013. The C++ Programming Language. Pearson Education.
- [20] The Intel oneAPI DPC++ compiler. [online] <https://github.com/intel/llvm>
- [21] 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).
- [22] Payne JL, Sinnott-Armstrong NA, Moore JH. Exploiting graphics processing units for computational biology and bioinformatics. *Interdiscip Sci*. 2010 Sep;2(3):213-20. doi: 10.1007/s12539-010-0002-4. Epub 2010 Jul 25. PMID: 20658333; PMCID: PMC2910913.
- [23] Niel, C., Sinoquet, C., Dina, C. and Rocheleau, G., 2015. A survey about methods dedicated to epistasis detection. *Frontiers in Genetics*, 6, p.285.
- [24] Nobre, R., Santander-Jiménez, S., Sousa, L. and Ilic, A., 2020, May. Accelerating 3-way Epistasis Detection with CPU+ GPU processing. In Workshop on Job Scheduling Strategies for Parallel Processing (pp. 106-126). Springer, Cham.
- [25] Lamb, J., Crawford, E.D., Peck, D., Modell, J.W., Blat, I.C., Wrobel, M.J., Lerner, J., Brunet, J.P., Subramanian, A., Ross, K.N. and Reich, M., 2006. The Connectivity Map: using gene-expression signatures to connect small molecules, genes, and disease. *science*, 313(5795), pp.1929-1935.
- [26] Zhang, S.D. and Gant, T.W., 2009. sscMap: an extensible Java application for connecting small-molecule drugs using gene-expression signatures. *BMC bioinformatics*, 10(1), pp.1-4.
- [27] McArt DG, Bankhead P, Dunne PD, Salto-Tellez M, Hamilton P, Zhang SD. cudaMap: a GPU accelerated program for gene expression connectivity mapping. *BMC Bioinformatics*. 2013 Oct 11;14:305.
- [28] Karakoc, E., Cherkasov, A. and Sahinalp, S.C., 2006. Distance based algorithms for small biomolecule classification and structural similarity search. *Bioinformatics*, 22(14), pp.e243-e251.
- [29] Guan, J., Chen, M., Ye, C., Cai, J.J. and Ji, G., 2018. AEGS: identifying aberrantly expressed gene sets for differential variability analysis. *Bioinformatics*, 34(5), pp.881-883.
- [30] Tahiri, N., Fichet, B. and Makarenkov, V., 2022. Building alternative consensus trees and supertrees using k-means and Robinson and Foulds distance. *Bioinformatics*, 38(13), pp.3367-3376.
- [31] Heterogeneous computing benchmarks. [online] <https://github.com/zjinf/HeCBench>
- [32] 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).
- [33] Parallel thread execution ISA Version 7.8. [online] <https://docs.nvidia.com/cuda/parallel-thread-execution/index.html>
- [34] Farber, R., 2011. CUDA application design and development. Morgan Kaufmann (pp 85-108)
- [35] Krainiuk, M., Goli, M. and Pascuzzi, V.R., 2021, November. oneAPI Open-Source Math Library Interface. In 2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC) (pp. 22-32). IEEE.
- [36] The Intel oneAPI Math Kernel Library (oneMKL) Interfaces [online] <https://github.com/oneapi-src/oneMKL>
- [37] The NVIDIA CUDA Driver API reference manual. [online] https://docs.nvidia.com/cuda/pdf/CUDA_Driver_API.pdf