Evaluating Nonuniform Reduction in HIP and SYCL on GPUs

Zheming Jin
Oak Ridge National Laboratory
jinz@ornl.gov

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

***Abstract*—Motivated by maturing programming models and portability for heterogeneous computing, we describe the challenges posed by hardware architectures and programming models when migrating an optimized implementation of nonuniform reduction from CUDA to HIP and SYCL. We explain the migration experience, evaluate the performance of the reduction on GPU-based computing platforms, and provide feedback on improving portability for the development of the SYCL programming model.**

***Keywords*— Nonuniform reduction, programming model, heterogeneous computing**

I. Introduction

Nonuniform reductions filter out data based on its content on a per-element basis. Such reduction realizes data filtering through a method of stream compaction [1]. Data filtering is useful in collision detection [2], tree traversal [3], ray tracing [4], etc. The reductions can be considered as a form of load balancing because a filtered input range makes it easier to provide an equal workload for all workers in a processor [5].

Previous studies focus on the designs and optimizations of the parallel reduction on NVIDIA graphics-processing units (GPUs) using the CUDA programming language [1,5,6]. With the advancement of AMD GPUs and their adoption in facilities and datacenters for scientific computing [7,8], migrating the nonuniform reduction from CUDA to programming models targeting these computing platforms will promote an understanding of the challenges and opportunities of portability. GPU computing platforms are different in the details of vendors' hardware architectures and the software stacks, so vendor-specific programming libraries and languages have been addressing the differences. However, the commonalities among these programming models exist that several portable programming approaches allow for writing code that supports multiple target platforms [9].

In this study, we describe the experience of migrating an optimized implementation of the nonuniform reduction from CUDA to other programming models. More specifically, we explain our approach to addressing the challenges posed by the differences in portable programming models and hardware architectures. Then, we evaluate the performance of the reductions on NVIDIA and AMD GPUs. Finally, we explain the performance gap and suggest performance improvement for the reduction.

We have described the motivation of our work. The rest of the paper is organized as follows. Section II introduces the programming models in our study, the nonuniform reduction and its implementation. Section III describes the challenges posed by the differences in software/hardware and our migration paths to address them. Section IV presents the experimental results. Section V discusses related work, and Section VI concludes the paper.

II. Background

A. A brief introduciton to programming models

CUDA is a parallel computing platform and application programming interface (API) that allows software to use NVIDIA GPUs for general-purpose processing [10]. Because CUDA is a mature and widely used programming model, we will focus on an introduction to the programming models that progress to maturity in this section.

Heterogeneous-Computing Interface for Portability (HIP) is a C++ runtime application programming interface (API) and kernel language developed mainly for programs executing on AMD GPUs. It offers a C-style API and C++ kernel language. While CUDA and HIP are very similar in the naming of APIs, HIP is a strong subset of CUDA in terms of functionality. On the other hand, AMD and NVIDIA GPUs are not the same in microarchitecture [11], so HIP supports architecture- and language-specific features for AMD accelerators. A HIP program is compiled with the HIP compiler in ROCm, an open software platform composed of tools, libraries, models, compilers, and runtimes [12].

SYCL is a promising programming model that builds on the underlying concepts, portability, and efficiency of Open Computing Language while adding much of the ease of use and flexibility of single-source C++ [13]. The higher abstractions in SYCL, such as device selectors, buffers, atomic reference, significantly improve productivity and efficiency of writing a parallel program for heterogenous computing devices. The Intel DPC++ is a compiler implementation of the SYCL specification based on the open-source LLVM technology [14]. The compiler allows code reuse across different Intel hardware platforms. While the compiler is optimized with vendor-specific features, it offers support of running a SYCL program on AMD and NVIDIA GPUs [15].

Portability is an important aspect of SYCL. When a SYCL program executes on a variety of platforms, it is desirable that the program can achieve reasonable performance on these platforms. However, a language and compiler can only guarantee that they can make it a little easier for developers to achieve portability for an application [13]. Hence, it is worthwhile to investigate and improve portability using SYCL.

B. Nonuniform reduction and its implementation

Listing 1 shows the sequential execution of nonuniform reduction over "N" elements. When the value of an element does

```
j = 0
for ( i = 0; i < N; i++ )
  if input[i] is valid
    output[j] = input[i]
    j++
```

Listing 1. Sequential nonuniform reduction

```
// Phase 1 counts valid elements per thread block
parallel for each block in blocks[0..GS-1]
  count = 0
  for each input element e in the block
    if e is valid
      count++
  blocks[bid] = count

// Phase 2 computes offsets of these blocks
offsets[0..GS-1] = xscan over blocks[0..GS-1]

// Phase 3 reduces the input nonuniformly
parallel for each offset in offsets[0..GS-1]
  j = offsets[oid]
  for each input element e processed by a processor
    if e is valid
      output[j] = e
      j++
```

Listing 2. Parallel nonuniform reduction implemented with three phases described in [5]. "GS" is the total number of thread blocks allocated for the workload and "BS" the total number of threads in a thread block. Each thread holds the value of an input element. The "range[..]" notation indicates the number of input elements in the range. "xscan" indicates exclusive scan. "bid" stands for block index and "oid" offset index.

```
1 template <typename T, typename Predicate>
2 void computeBlockCounts(
3   const T*__restrict d_input,
4   int length,
5   int* __restrict d_BlockCounts,
6   Predicate predicate,
7   nd_item<1> &item)
8 {
9   int idx = item.get_global_id(0);
10  if(idx < length){
11    int pred = predicate(d_input[idx]);
12    int c = sycl::reduce_over_group(
                item.get_group(), pred,
                sycl::ext::oneapi::plus<>());
13    if (item.get_local_id(0) == 0) {
14      d_BlockCounts[item.get_group(0)] = c;
15    }
16  }
17 }
```

Listing 3. The SYCL kernel for the phase 1 of the reduction. The CUDA intrinsic "__syncthreads_count()" is mapped to the SYCL group reduction function. The indices of work-items and work-groups are queried with the methods of the SYCL class "nd_item".

not meet some condition (i.e., invalid), the element will be filtered out. Hence, only valid elements will be stored consecutively in the output. The challenge of the parallel reduction lies in the dependency of output location of each element on the validity of every element before it [5].

In [16], the author presented a highly optimized CUDA implementation of the reduction targeting an NVIDIA GPU. The method consists of three phases as shown in Listing 2. The first phase counts the number of valid elements per thread block. Then, the first thread in each thread block stores the result in the device memory. In the second phase, a prefix-sum operation [17] on the number of valid elements of each thread block is performed to produce a vector of offsets for all thread blocks. The prefix sum is implemented using the Thrust library for productivity [18]. The last phase reduces the input values with compaction and outputs the compacted values with the block offsets computed in the second phase. The implementation of the compaction is optimized with the CUDA intra-warp voting function [19], population count operation [19], and bit manipulation to achieve efficiency and performance on an NVIDIA GPU.

III. Migration Challenges from CUDA to HIP and SYCL

Previous studies described the migration of benchmarks and applications from CUDA to HIP and/or SYCL [20,21,22,23,24]. For the reduction, we will focus on the challenges of migrating the CUDA implementation and how they are addressed. Before diving into the implementation details, we should clarify the terminology commonly used in the kernel languages. Thread, thread block, and warp in CUDA correspond to thread, thread block, wavefront in HIP, respectively; they correspond to work-item, work-group, and sub-group in SYCL, respectively. In the SYCL specification, work-items have access to group functions that implement common communication routines and parallel patterns such as reduction and scan [25]. A sub-group refers to subsets of work-items in a work-group. The size of a sub-group can be specified at compile-time as a kernel attribute, but it must be compatible with the sizes supported by a target device. Like CUDA warp-level optimizations, the execution of work-items at the granularity of sub-group (i.e., close to hardware) may achieve higher level of performance across GPU platforms.

The CUDA kernel for the first phase of the reduction calls a barrier intrinsic function "__syncthreads_count(p)" that counts the number of non-zero predicates (p) of all threads in a thread block and synchronizes the operations to return the result to all threads [19]. Migrating the intrinsic to HIP is straightforward as it is natively supported by the language. In SYCL, the intrinsic is mapped to the "reduce_over_group()" function provided by the group algorithms library [25]. Since there are no code changes from CUDA to HIP for the first phase, we list the SYCL kernel in Listing 3.

The input elements and output results are stored on GPU device memory before the kernel starts executing. In the kernel, the predicate is true when the value of an input element is positive (L11). In a real application, a predicate's value may be computed by comparing each element's value with a threshold. The SYCL group function (L12) is specialized to sum up the number of valid elements in a work-group. Finally, the first work-items (L13) of all work-groups store the results of the group reduce function in "d_BlockCounts" at the locations corresponding to the work-group indices (L14). Besides reads from and writes to device memory by work-items in each work-group, most of the kernel execution time is spent on counting the number of valid elements of all work-items in each work-group and synchronizing the operations.

In the second phase of the reduction, an exclusive scan over the valid predicates counted in the first phase produces a vector

```
auto policy =
   oneapi::dpl::execution::make_device_policy(q);
oneapi::dpl::exclusive_scan(
     policy,
     d_BlocksCount,
     d_BlocksCount + numBlocks,
     d_BlocksOffset,
     (T)0);
```

Listing 4: The library-based approach for exclusive scan in the SYCL kernel

of block offsets (d_BlocksOffset). Both the CUDA and HIP programs call the "thrust::exclusive_scan()" function in the Thrust library for the scan operation. In the SYCL program, the Thrust scan is migrated with the Intel oneAPI DPC++ Library (oneDPL) [26].

Listing 4 shows the oneDPL exclusive scan function. Both the Thrust and oneDPL functions require the beginning and end of an input sequence, the beginning of an output sequence, and an appropriate initial value. However, the oneDPL scan requires an execution policy for specifying that the parallel algorithm's execution may be parallelized on a target device. Additionally, the scan function depends on the Intel performance library for thread building block (oneTBB) [27]. Though we no longer need to implement scan operation from scratch, the comparison shows that migrating the scan function from CUDA to SYCL depends on the support of multiple libraries and SYCL-specific scan function. Apparently, the performance of the scan depends on the performance of the library implementation.

The last phase of the reduction computes the predicate offset for each thread in a thread block by dividing a thread block into warps. Migrating the CUDA kernel for this phase to HIP is less straightforward than the process in the first two phases due to the architectural differences.

Listing 5 shows the kernel in HIP. We will explain the differences between the CUDA and HIP kernels when going through the operations in the kernel. The warp and thread indices within a warp are computed on L13 and L14, respectively. When producing a thread mask for active threads in a warp (L15), we find the result of shifting right the default 32-bit thread mask (all ones) with a shift amount of 32 is zero in CUDA, yet the result is unchanged in HIP for the 64-bit mask (L15). The intra-warp voting function "__ballot()"produces a bitmask whose i^{th} bit is set when the value of the predicate held by the i^{th} thread's true (L16). For the NVIDIA GPU architecture, there are 32 parallel threads in a warp. In contrast, the number of parallel threads is 64 in a wavefront (warp) for the AMD MI-series GPUs architecture [28]. Hence, the "__ballot()" returns a 64-bit result from the predicate evaluation of 64 threads. Then, it is combined with the population count for an efficient implementation of Boolean reduction. The appropriate function for population count is called to count the number of non-zero predicates in a 64-bit number (L17). The last thread in each warp stores the total number of valid predicates in a warp in a shared memory (L18, L19). L20 synchronizes memory writes of all warps in a thread block. The number of warps (numWarps) is the size of a thread block divided by the warp size (L21). It is assumed that the number of warps in a thread block is no more than the warp size. Then, threads of size "numWarps" in the first warp perform an exclusive prefix sum to produce output offset for each warp in a thread block. A naïve way to compute the sum will loop over the values in the shared memory with the trip count equal to "numWarps". Computing the accumulative warp offsets is optimized with a binary-manipulation loop (L22 – L30) where the trip count equals the number of bits that can represent the maximum offset value for a warp. When the warp size is 64, the trip count increases from 5 to 6 (L24). When the value of a predicate held by a thread is true (L31), the value is stored in the destination address computed with thread offset in a warp, warp offset in a block, and block offset in a grid (L32).

The portability feature of SYCL does not necessarily mean that a single SYCL kernel for the last phase can execute correctly across different GPUs. The kernel needs to consider

```
   #define warpSize (64)
   #define FULL_MASK 0xffffffffffffffffUL
 1 template <typename T, typename Predicate>
 2 __global__ void compactK(
 3    const T*__restrict__ d_input,
 4    int length,
 5          T*__restrict__ d_output,
 6    const int*__restrict__ d_BlocksOffset,
 7    Predicate predicate)
 8 {
 9   extern __shared__ int warpTotals[];
10   int idx = threadIdx.x + blockIdx.x * blockDim.x;
11   if(idx < length){
12     int pred = predicate(d_input[idx]);
13     int w_i = threadIdx.x / warpSize;
14     int w_l = idx % warpSize;
15     size_t t_m = (w_l == 0) ? 0 :
                   (FULL_MASK >> (warpSize-w_l));
16     size_t b = __ballot(pred) & t_m;
17     unsigned int t_u = __popcll(b);
18     if(w_l == warpSize-1)
19       warpTotals[w_i]=t_u+pred;
20     __syncthreads();
21     int numWarps = blockDim.x / warpSize;
22     if(w_i==0 && w_l<numWarps){
23       int w_i_u=0;
24       for(int j=0;j<=6;j++){
25         int b_j =__ballot( warpTotals[w_l] &
                    pow2i(j) );
26         w_i_u += (__popc(b_j & (t_m &
                    0xFFFFFFFF))) << j;
27       }
28       warpTotals[w_l]=w_i_u;
29     }
30     __syncthreads();
31     if(pred){
32       d_output[t_u + warpTotals[w_i]+
              d_BlocksOffset[blockIdx.x]]=
         d_input[idx];
33     } } }
```

Listing 5. The HIP kernel for the phase 3 of the reduction. The wavefront size is 64 for the kernel.

```
1 auto sg = item.get_sub_group();
2 size_t b = sycl::reduce_over_group(sg,
          pred ? (1UL << sg.get_local_linear_id())
          : 0, sycl::ext::oneapi::plus<>());
```

Listing 6: Map the CUDA "__ballot()" to the SYCL group reduction function over a sub-group. The required sub-group size can be set at compile time. It is 64 for the kernel targeting an AMD GPU.

```
1 int t = 0;
2 if (w_i == 0 && w_l < numWarps){
3   t = warpTotals[w_l];
4 }
5 if (w_i == 0) {
6    for(int j=0;j<=6;j++){
7      unsigned int b_j = reduce_over_group(sg,
         (t & pow2i(j)) ?
        (0x1 << sg.get_local_linear_id()) : 0,
         ext::oneapi::plus<>());
8      w_i_u += sycl::popcount(b_j & t_m32) << j;
9    }
10 }
```

Listing 7: Bit manipulation in the SYCL kernel. The required sub-group size can be set at compile time. It is 64 for the kernel targeting an AMD GPU. All work-items in a sub-group execute the group reduction function; otherwise, the result of the reduction is incorrect.

the architectural differences between the AMD and NVIDIA GPUs. The "__ballot()" function is converted to the SYCL group function as shown in Listing 6. Instead of reducing over the constituent work-group as in Listing 3, the function performs a reduction over a sub-group in which all work-item participate in computing the values of predicates. The sub-group size is either 32 or 64, which is specified with the kernel attribute "reqd_sub_group_size()" when launching the SYCL kernel on the host.

We find that the result using the SYCL group reduction function does not match that of the CUDA or HIP intra-warp voting function when computing the accumulative warp offsets with a binary-manipulation loop. The SYCL group function expects that the number of work-items executing the function equals the sub-group size. However, the actual number of active work-items (i.e., numWarps) may be less than the sub-group size. The issue can be addressed by allowing all work-items in a sub-group to execute the function shown in Listing 7. We suggest that the SYCL specification clarify the behavior of the group reduction operation when the number of active work-items is less than the sub-group size.

IV. Experiments

A. Setup

We evaluate the performance of the reduction on three GPU-based platforms. The first platform (P1) is NVIDIA Jetson AGX Xavier that consists of an ARM v8.2 CPU and an integrated Volta GPU. The second platform (P2) contains an Intel Xeon E5-2698 v4 CPU and an NVIDIA V100 DGXS GPU. The third platform (P3) contains an AMD EPYC 7272 CPU and an AMD MI100 GPU. The major specifications of the GPUs are listed in Table I. The device-to-device (D2D) memory bandwidths are measured with the bandwidth tests in CUDA and HIP. The CUDA programs are compiled with the JetPack v5.0.1 and HPC SDK v22.7 on P1 and P2, respectively. The HIP program is compiled with the ROCm v4.5.2. The SYCL compiler with NVIDIA and AMD GPU support is built from the latest release, version 2022-6. The number of input elements (problem size) for reduction ranges from 2^{20} to 2^{29} and the size of each element is four bytes. It should be noted that the number of elements is not necessarily a power of two for the reduction. The size of a thread block is a power of two ranging from 64 to 1024. The performance metrics are throughput and efficiency. We define the *throughput* as Giga elements reduced per second (G/s) and the *efficiency* as effective utilization as measured by a comparison with device-to-device memory bandwidth:

$$Throughput = \frac{number\ of\ input\ elements}{average\ reduction\ time}$$

$$Efficiency = \frac{maximum\ throughput\ in\ bytes}{D2D\ memory\ bandwidth}$$

TABLE I. DEVICE SPECIFICATIONS (D2D: DEVICE-TO-DEVICE)

GPU Specification	**Xavier**	**V100**	**MI100**
Global memory (GB)	16	32	32
GPU clock rate (MHz)	1377	1530	1502
Memory clock rate (MHz)	1377	877	1200
Number of cores	512	5120	7680
L2 Cache (MB)	0.5	6	8
Peak memory bandwidth (GB / s)	137	898	1228
D2D memory bandwidth (GB / s)	62	732	770

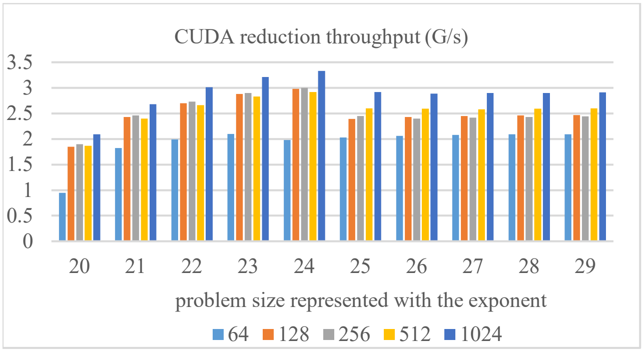

Fig. 1a. Performance of the CUDA reduction with respect to the problem and thread block sizes on P1

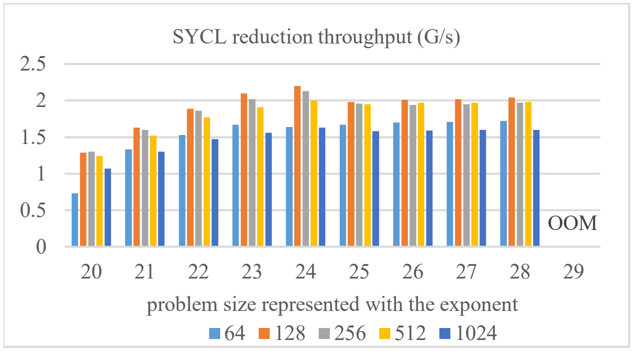

Fig. 1b. Performance of the SYCL reduction with respect to the problem and thread block sizes on P1. (OOM: out-of-memory)

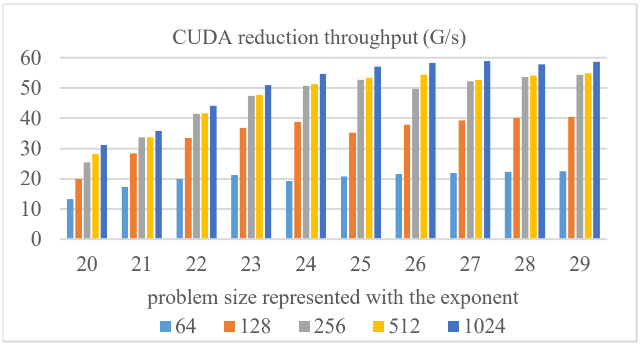

Fig. 2a. Performance of the CUDA reduction with respect to the problem and thread block sizes on P2

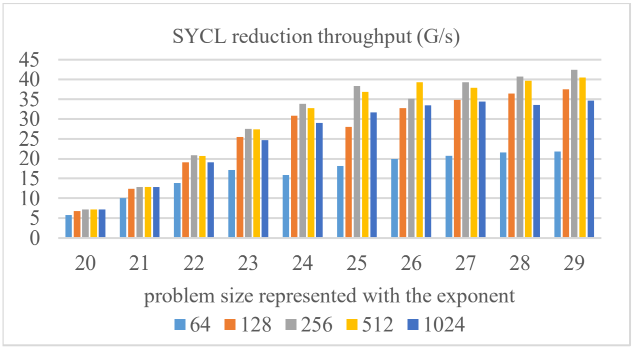

Fig. 2b. Performance of the CUDA reduction with respect to the problem and thread block sizes on P2

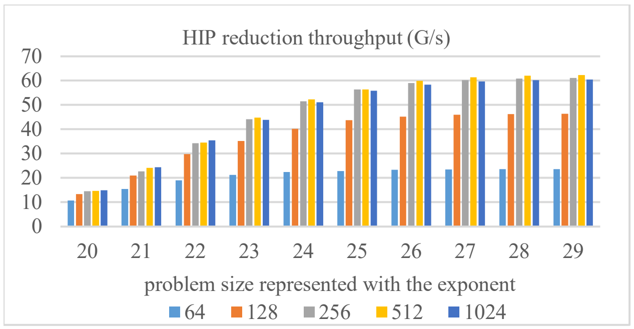

Fig. 3a. Performance of the HIP reduction with respect to the problem and thread block sizes on P3

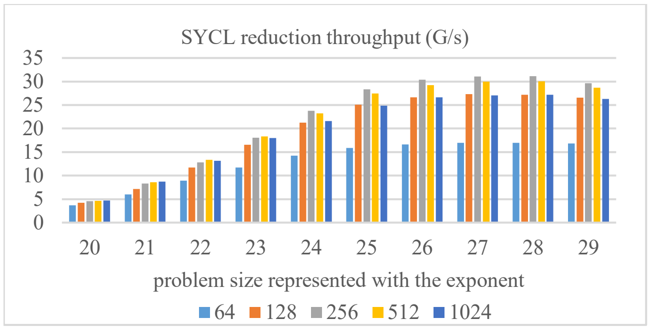

Fig. 3b. Performance of the SYCL reduction with respect to the problem and thread block sizes on P3

We repeat the three phases of the reduction 1000 times to obtain average reduction time. The GPU results are verified on each host.

B. *Results*

Our experiments show that the reduction throughput reaches 50% or more of the maximum when the problem size is 2^{23}. Memory bandwidth is underutilized when the problem size is small for parallel processing. The performance of the reduction is sensitive and tunable with the thread block size across the problem sizes.

Fig. 1a and Fig. 1b show the reduction performance on P1 using CUDA and SYCL, respectively. The highest CUDA and SYCL reduction throughputs are approximately 3.33 G/s and 2.2 G/s, respectively. The SYCL runtime reports an out-of-memory error when allocating memory for the largest problem size. Fig. 2a and Fig. 2b show the reduction throughput on P2 using CUDA and SYCL, respectively. The highest CUDA and SYCL reduction throughputs are approximately 58.8 G/s and 42.2 G/s, respectively. Fig. 3a and Fig. 3b show the reduction performance on P3 using HIP and SYCL, respectively. The highest HIP and SYCL reduction throughputs are approximately 62.3 G/s and 31.1 G/s, respectively. Hence, the maximum throughputs of the SYCL reduction with CUDA and HIP support are 1.5X, 1.39X and 2.0X lower than those of the CUDA and HIP reductions on the three platforms, respectively. The maximum CUDA reduction throughput is reached when the thread block size is 1024 on P1 and P2. However, we observe approximately 13% to 20% performance drop when increasing the work-group size from 256 to 1024 using SYCL. On the AMD GPU, the maximum HIP and SYCL throughputs are reached when the block sizes are 512 and 256, respectively. Doubling the optimal size incurs a maximum of 4% performance drop.

TABLE II. THE REDUCTION EFFICIENCY ON THE THREE PLATFORMS

Efficiency (%)	CUDA	HIP	SYCL
P1	21.48	N/A	14.19
P2	32.13	N/A	23.06
P3	N/A	32.36	16.15

TABLE III. EXECUTION TIME OF THE THREE PHASES OF THE REDUCTION

Time (us)	P1	P2	P3
HIP	4338	58.2	4385
SYCL	6390	264	9865

Table II shows the reduction efficiency on the three platforms. We focus on the performance portability of the SYCL programming model on NVIDIA and AMD GPUs. For a memory-bound kernel, the efficiency of the CUDA and HIP reductions are almost identical on P2 and P3. P1 is a platform for embedding computing while P2 and P3 are high-performance computing (HPC) platforms. The HPC platforms are more efficient in terms of throughput than the embedded computing platform for the reduction. There is a large space of improving the efficiency of the SYCL reductions on these platforms.

An integrated GPU is not designed to outperform a discrete GPU due to the power, area, and thermal constrains. Comparing the reduction performance on the discrete GPUs shows that the maximum HIP throughput on P3 is about 1.06X higher than the maximum CUDA throughput on P2. However, the SYCL throughput on P3 is about 1.36X lower than that on P2. Hence, we profile the three phases of the reductions on the AMD GPU when the problem size is 2^{29} and the block size is 256.

Table III shows the HIP and SYCL execution time in microseconds (us) of the three phases. In the first phase (P1), the implementation of the barrier intrinsic "__syncthreads_count()" in the HIP kernel is more efficient than the implementation of "reduce_over_group()" in SYCL for counting the number of non-zero predicates in a wavefront. In the second phase, the implementation of the exclusive scan in the HIP Thrust is more efficient than that in the oneDPL. In the third phase, the HIP compiler can convert the bit manipulation executed by a single wavefront to operations over a vector register. The SYCL compiler is not able to reduce the bit operations using a vector register. Comparing the assembly codes of the HIP and SYCL kernels also shows that the HIP compiler can generate more efficient instructions in terms of the total instruction length in bytes and the number of allocated scalar and vector general-purpose registers.

V. RELATED WORK

Our experimental results show that migrating the non-uniform reduction from CUDA to SYCL is more challenging than the process from CUDA to HIP. Hence, we prefer a discussion of previous studies on functional and performance portability [20-24] to the designs and optimizations of the CUDA nonuniform reduction [5,6,16,19]. In [20], the authors evaluate the performance of benchmarks and mini-apps having both SYCL and CUDA implementations on a NVIDIA V100 GPU. They find many of the performance differences are due to the ordering and choices of memory accesses. Our evaluation shows that the performance differences are caused by the implementations of the SYCL runtime and library for the reduction functions. In [21], the authors evaluate the performance of a GPU accelerated sequence alignment algorithm across vendors' GPUs using CUDA, HIP and SYCL. They find that migrating the highly optimized CUDA kernels to SYCL requires significant code changes. The SYCL implementation is 2X slower than the CUDA implementation on the target devices. Putting aside the differences in the language design of programming models, we will continue working with the SYCL community for optimizing the SYCL compiler to improve performance portability. In [22], the authors describe their experiences of migrating NAMD, a large molecular dynamics software application, from CUDA to SYCL. While porting most CUDA kernels in the application is straightforward, they take the library-based approach for migrating the reduction, scan, sort, and other operations in the CUDA application. Our experiment shows that the oneDPL library is about 4.5X lower than the Thrust library in terms of the scan performance. In [23], the authors evaluate the performance of a machine-learning application with SYCL and CUDA on multiple NVIDIA GPUs. They point out that performance portability has not yet been fully achieved by any SYCL implementations. CUDA's mature development environment and its variety of libraries can speed up the development process on NVIDIA platforms. As HIP and SYCL are maturing and being deployed in facilities, developers are encouraged to find gaps between CUDA and other programming models for improving performance portability. In [24], the author describes the experience of migrating a graph application from CUDA to SYCL. The CUDA and SYCL application are comparable in kernel execution time on the NVIDIA GPUs, but certain CUDA device property, math function, and warp primitive were not fully supported by SYCL built-in functions. Our experiment shows that the SYCL group reduction functions, to which the warp-level primitives are mapped, have become a performance bottleneck for the reduction. As a summary of related work, portability depends on the characteristics of an application, the optimizations applied to the application, the maturity of the toolchains and libraries, and the expressiveness of the programming models.

VI. Conclusion

We successfully migrate the kernels in a nonuniform reduction from CUDA to HIP and SYCL by addressing the challenges posed by differences in programing models and GPU architectures. This requires a good understanding of the impact of the architectural differences upon the programming models. Portability is one of the key features in SYCL, but we still need to address the architectural differences in the SYCL kernels. For the reduction kernels, the migration from CUDA to SYCL is more challenging than the relatively straightforward process from CUDA to HIP. With the development of the SYCL ecosystem, we expect that the SYCL-specific group algorithm library and data-parallel library will be improved to narrow the performance gap between the SYCL and CUDA/HIP implementations. In the future work, we are interested in evaluating the reduction on an Intel data-center GPU.

T

Acknowledgment

We appreciate the reviewers' comments and suggestions. This research used resources of the Experimental Computing Lab at Oak Ridge National Laboratory. This research was supported by the US Department of Energy Advanced Scientific Computing Research program under Contract No. DE-AC05-00OR22725.

References

[1] Pharr, M. and Fernando, R., 2005. GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation (GPU Gems). Addison-Wesley Professional.

[2] Greß, A., Guthe, M. and Klein, R., 2006, September. GPU-based collision detection for deformable parameterized surfaces. In Computer Graphics Forum (Vol. 25, No. 3, pp. 497-506). Oxford, UK and Boston, USA: Blackwell Publishing, Inc.

[3] Lauterbach, C., Garland, M., Sengupta, S., Luebke, D. and Manocha, D., 2009, April. Fast BVH construction on GPUs. In Computer Graphics Forum (Vol. 28, No. 2, pp. 375-384). Oxford, UK: Blackwell Publishing Ltd.

[4] Wald, I., Gribble, C.P., Boulos, S. and Kensler, A., 2007. SIMD ray stream tracing-simd ray traversal with generalized ray packets and on-the-fly re-ordering. Informe Técnico, SCI Institute, 2.

[5] Billeter, M., Olsson, O. and Assarsson, U., 2009, August. Efficient stream compaction on wide SIMD many-core architectures. In Proceedings of the conference on high performance graphics 2009 (pp. 159-166).

[6] Rego, V., Sang, J. and Yu, C., 2016, November. A fast hybrid approach for stream compaction on GPUs. In 2016 Fourth International Symposium on Computing and Networking (CANDAR) (pp. 476-482). IEEE.

[7] Schulte, M.J., Ignatowski, M., Loh, G.H., Beckmann, B.M., Brantley, W.C., Gurumurthi, S., Jayasena, N., Paul, I., Reinhardt, S.K. and Rodgers, G., 2015. Achieving exascale capabilities through heterogeneous computing. IEEE Micro, 35(4), pp.26-36.

[8] Germaschewski, K., Allen, B., Dannert, T., Hrywniak, M., Donaghy, J., Merlo, G., Ethier, S., D'Azevedo, E., Jenko, F. and Bhattacharjee, A., 2021. Toward exascale whole-device modeling of fusion devices: Porting the GENE gyrokinetic microturbulence code to GPU. Physics of Plasmas, 28(6), p.062501.

[9] Portability Across DOE Office of Science HPC Facilities. [online] https://performanceportability.org/

[10] 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.

[11] Gutierrez, A., Beckmann, B.M., Dutu, 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.

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

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

[14] Intel oneAPI DPC++ compiler. [online] Available: https://github.com/intel/llvm

[15] Spataro, D., 2015. Stream compaction on GPU. [online] Available: http://www.davidespataro.it/cuda-stream-compaction-efficient-implementation/

[16] Hillis, W.D. and Steele Jr, G.L., 1986. Data parallel algorithms. Communications of the ACM, 29(12), pp.1170-1183.

[17] Bell, N. and Hoberock, J., 2012. Thrust: A productivity-oriented library for CUDA. In GPU computing gems Jade edition (pp. 359-371). Morgan Kaufmann.

[18] Harris, M. and Garland, M., 2012. Optimizing parallel prefix operations for the Fermi architecture. In GPU Computing Gems Jade Edition (pp. 29-38). Morgan Kaufmann.

[19] 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).

[20] 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.

[21] David J Hardy, Jaemin Choi, Wei Jiang, and Emad Tajkhorshid. 2022. Experiences Porting NAMD to the Data Parallel C++ Programming Model. In International Workshop on OpenCL (IWOCL'22). Association for Computing Machinery, New York, NY, USA, Article 15, 1–5.

[22] 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.

[23] Jin, Zheming. 2022. Experience of Migrating Parallel Graph Coloring from CUDA to SYCL. United States. https://www.osti.gov/servlets/purl/1864412.

[24] The SYCL 2020 Specification (revision 5). [online] https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html.

[25] Intel oneAPI DPC++ Library. [online] Available: https://github.com/oneapi-src/oneDPL

[26] Intel oneAPI Threading Building Blocks. [online] Available: https://github.com/oneapi-src/oneTBB

[27] AMD Instinct MI100 Instruction Set Architecture. [online] https://developer.amd.com/wp-content/resources/CDNA1_Shader_ISA_14December2020.pdf