# Evaluating the Performance of Integer Sum Reduction in SYCL on GPUs

Zheming Jin and Jeffrey Vetter
Computer Science and Mathematics
Oak Ridge National Laboratory
jinz@ornl.gov

## ABSTRACT

SYCL is a promising programming model for heterogeneous computing—allowing a single-source code to target devices from multiple vendors. One significant task performed on these accelerators is a primitive operation for integer sum reduction. This paper presents several SYCL implementations of integer sum reduction—using atomic functions, shared local memory, vectorized memory accesses and parameterized workload sizes—to compare the performance and maturity of SYCL against open-source vendor-specific implementations of the same reduction. For a sufficiently large number of integers, tuning the parameters of our SYCL implementations achieves 1.4X speedup over the open-source implementations on an Intel UHD630 integrated GPU. The SYCL reduction is 3% faster than the templated reduction in Thrust, and 0.3% faster than the device reduction in CUB on an Nvidia P100 GPU. The SYCL reduction is 1.9% faster than the templated reduction in Thrust, and 0.4% faster than the device reduction in CUB on an Nvidia V100 GPU.

## CCS CONCEPTS

**• Computing methodologies → Parallel computing methodologies → Parallel algorithm; • Software and its engineering → Parallel programming language.**

## KEYWORDS

Reduction, OpenCL, CUDA, SYCL, GPGPU

**ACM Reference format:**

FirstName Surname, FirstName Surname and FirstName Surname. 2018. Insert Your Title Here: Insert Subtitle Here. In *Proceedings of ACM Woodstock conference (WOODSTOCK'18). ACM, New York, NY, USA, 2 pages.* https://doi.org/10.1145/1234567890

## 1 Introduction

With the support of major graphics hardware vendors as well as personal computer vendors interested in offloading computations [1], Open Computing Language (OpenCL) is an open standard maintained by the Khronos group. OpenCL offers programming portability across a wide range of software and hardware for graphics processing units (GPUs), modern central processing units (CPUs), and other accelerators [2]. As opposed to the OpenCL programming model in which host and device codes are written in two languages [3], the promising SYCL standard specifies a cross-platform abstraction layer that enables programming of a heterogeneous computing system using standard C++ [4]. It can combine host and device codes for an application in a type-safe way to improve development productivity. The goals of the single-source programming model are to improve programming productivity and performance portability [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15].

Reduction is a primitive operation in parallel computing. In this paper, we evaluate parallel implementations of the integer sum reduction in SYCL on GPUs. Specifically, we explain our implementations of the reduction using atomic functions (atomics), shared local memory, vectorized memory accesses, and parameterized workload sizes. Then, we evaluate its performance with respect to tunable parameters on an Intel integrated GPU and Nvidia discrete GPUs. In addition, we evaluate major open-source implementations of the reduction for performance comparison. The results show that it is important to tune work-group size, vector width, and workload size to achieve the optimal performance for the sum reduction on a target device. By tuning the parameters, we can achieve 1.4X performance speedup over the open-source implementations on an Intel integrated GPU. Compared with the open-source vendor-specific implementations of the reduction, our SYCL implementations with parameter tuning are 0.3% to 3% faster than the templated and device reductions on Nvidia GPUs. While the templated and device reductions are mature for parallel reduction on an Nvidia GPU, there is a potential of improving the performance of the framework-agnostic reduction class supported by SYCL compilers.

The rest of the paper is organized as follows. In Section II, we contrast the SYCL and OpenCL programming models from an application perspective. Then, we give an overview of SYCL with CUDA support, and describe the scope of the reduction in our study. Section III explains the SYCL kernels and evaluates the performance of the reduction implementations on the GPUs. Section IV summarizes related work, and Section V concludes the paper.

## 2 Background

### 2.1 SYCL Overview

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

**Table 1: Mapping from OpenCL to SYCL**

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

A SYCL application is logically structured in three scopes: a kernel scope, a command-group scope, and an application scope. A kernel scope specifies a kernel, typically a compute-intensive function, that will be offloaded to a device (e.g., a GPU) for acceleration. A command-group scope specifies a unit of work that comprises of a kernel function and accessors to data allocated in global, local, or constant memory address space [1]. An application scope specifies all other code beyond a command-group scope. A SYCL kernel function may be defined by the body of a lambda function, by a function object, or by a binary generated from an OpenCL kernel string. Although an OpenCL kernel is interoperable in the SYCL programming model, we implement kernels functions using lambda.

Table 1 shows the major differences between an OpenCL application and a SYCL application. Searching hardware platforms and creating context for each platform's device in OpenCL can be simplified to the instantiation of a device selector class in SYCL. A selector searches a device of a user's provided preference (e.g., GPU) at runtime. The SYCL queue class encapsulates a command queue for scheduling kernels on a device. A kernel function in SYCL can be invoked as a lambda function. It is grouped into a command group object. Then, it is submitted to execution via a command queue. Hence, steps 6 to 10 in OpenCL are mapped to the definition of a lambda function and submission of its command group to a SYCL queue. Data transfers between a host and a device can be implicitly realized by SYCL accessors. The SYCL event class deals with event handling. Releasing memory resources, such as a queue, a program, a kernel, and memory objects, can be handled by the SYCL runtime implicitly. Compared to the number of steps taken for an OpenCL application, SYCL reduces the number of programming steps by half with the high-level abstraction.

## 2.2 SYCL with CUDA Support

Nvidia CUDA [ 16 ], which was introduced in 2007, has successfully enabled the use of a GPU as a programmable general-purpose computing device. However, CUDA is a proprietary programming model for Nvidia GPUs. In contrast to CUDA, the OpenCL application-programming interface (API) is a lower-level abstraction compared to the commonly used CUDA API, thus requiring more time and effort to develop an OpenCL host program for the management of device, memory, and kernel execution.

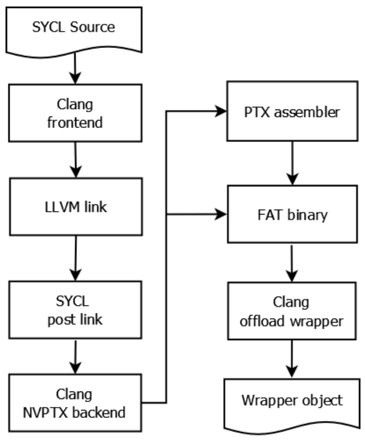

**Figure 1: CUDA target processing in the SYCL compiler [22]**

Acknowledging CUDA's established presence in high-performance computing, researchers have been striving for a portability-enhancing path for a wider set of platforms [17, 18, 19, 20]. SYCL with CUDA support is built upon the LLVM compiler framework [21]. The experimental CUDA support is publicly available in the Intel branch of the LLVM repository [22].

Figure 1 shows the flow that enables a SYCL program to execute on an Nvidia GPU. The details of the compiler infrastructure are described in [23]. The NVPTX backend, which generates a machine model and low-level virtual instructions (PTX) [ 24 ] for a source program, allows for a better understanding of the compiler optimizations applied to a source program.

## 2.3 Sum Reduction

Sum reduction is a primitive operation in parallel computing. The scope of our study is the unsegmented form of sum reduction. Taking a binary associative operator "+" and an array of "M" numbers as inputs, the reduction returns as output one value. Here, input values, output sum, and "M" are integers. Listing 1 shows the sequential integer sum reduction as a reference.

```
int numbers[M];
int sum = 0;
for ( int i = 0; i < M; i++ )
  sum += numbers[i];
```

**Listing 1: The sequential integer sum reduction in C**

For the integer sum reduction, its operations can be done in any order due to the associativity and commutativity of integer additions. Hence, we can divide the reduction into independent partial sums, compute each partial sum in an arbitrary order, and produce a result by combining these partial sums. The idea can be generalized to reductions on vectors of arbitrary size.

## 3 SYCL Implementations of the Sum Reduction

Our implementations are based on the idea that the integer sum reduction can be divided into independent partial sums, which can be computed in an arbitrary order to produce a result.

### 3.1 Reduction with Shared Local Memory

Listing 2 shows the reduction in which partial sums are computed using a shared local memory [1]. For clarity of description, we omit the namespace "cl::sycl::" required for accessing SYCL classes, methods, variables, etc. The "parallel_for" member function of the SYCL handler class provides an interface to define and invoke a SYCL kernel function in a command group (cg). The "nd_item" class encapsulates information to identify local and global identifiers of work-items specified in a three-dimensional space in a work-group. In the kernel, "sum" is a SYCL accessor to a shared local memory for storing partial sum computed by all work-items in a work-group; "input" and "output" are accessors to global memories for integers and the sum of integers, respectively. The first work-item in each work-group resets the sum. Then, a barrier synchronizes all work-items in a work-group to wait for the initialization of the local sum. After atomic additions have been performed by all work-items in a work-group, the last work-item in each group atomically adds these partial sums to the output. The global work size ("gws") is equal to the number of integers to sum ("M") while the local work size or work-group size ("lws") is a tunable parameter.

```
cgh.parallel_for<class reduce>(
  nd_range<1>(gws, lws),[=](nd_item<1> item) {
  int gid = item.get_global_id(0);
  int lid = item.get_local_id(0);
  int WGS = item.get_local_size(0);
  if (lid == 0) sum[0].store(0);
  item.barrier(access::fence_space::local_space);
  atomic_fetch_add(sum[0], input[gid]);
  item.barrier(access::fence_space::local_space);
  if (lid == WGS-1) {
    int partial_sum = atomic_load(sum[0]);
    atomic_fetch_add(out[0], partial_sum);
  }
});
```

**Listing 2: The SYCL kernel scope of the hierarchical reduction over global and shared local memories**

### 3.2 Reduction with Vectorized Memory Accesses

We attempt to improve the efficiency of global memory accesses with vectorized memory accesses. Listing 3 shows the kernel in SYCL-pseudocode with the number of vector lanes "N", which is a power of two, ranging from 1 to a maximum value of 16. The kernel shown in Listing 2 can be considered as a specific case of this kernel where "N" equals one. Using a SYCL vector class and its method, each work-item fetches "N" consecutive data from global memory as a vector. Then, it sums up the contents of the vector elements and stores the result in an intermediate variable "r". After atomic additions have been performed by all work-items in a work-group, the last work-item in each group atomically adds the partial sum to the output. The vectorized memory loads reduce the global work size by a factor of "N". The number of atomics over a shared local memory are also reduced by a factor of "N".

```
cgh.parallel_for<class reduce>(
  nd_range<1>(gws, lws), [=](nd_item<1> item) {
  vec<int, N> vi;
  int gid = item.get_global_id(0);
  int lid = item.get_local_id(0);
  int WGS = item.get_local_size(0);
  vi.load(gid, input.get_pointer());
  int r = vi.s0() + vi.s1() + … + vi.sN-1();
  if (lid == 0) sum[0].store(0);
  item.barrier(access::fence_space::local_space);
  atomic_fetch_add(sum[0], r);
  item.barrier(access::fence_space::local_space);
  if (lid == WGS-1) {
    int partial_sum = atomic_load(sum[0]);
    atomic_fetch_add(out[0], partial_sum);
  }
});
```

**Listing 3: The SYCL kernel scope of the hierarchical reduction with vectorized memory accesses**

### 3.3 Reduction with Parameterized Workload Sizes

As an alternative to vectorized memory accesses, we can increase the workload assigned to each work-item. As shown in Listing 4, each work-item in a work-group is assigned the workload of accumulating the numbers read from memory addresses "i", "i+WGS", "i+2×WGS",…, "i+L×WGS", where "i", "WGS", and "L" are the memory address for the first integer accessed by each work-item in a work-group, the work-group size, and the workload size assigned to each work-item, respectively. The sum of integers is stored in an intermediate variable "r". After atomic additions have been performed by all work-items in a work-group, the last work-item in each group atomically adds these partial sums to the output. In contrast to "N" which is capped at 16, "L" is a power of two with the constraint that the global work size (i.e., "M/L") is no less than the local work size.

```
cgh.parallel_for<class reduce>(
  nd_range<1>(gws, lws), [=](nd_item<1> item) {
  int gid = item.get_global_id(0);
  int lid = item.get_local_id(0);
  int blk = item.get_group(0);
  int WGS = item.get_local_size(0);
  if (lid == 0) sum[0].store(0);
  item.barrier(access::fence_space::local_space);
  int start = blk * WGS * L + lid;
  int end = (blk+1) * WGS * L;
  int r = 0;
  for (int i = start; i < end; i = i + WGS)
    r += input[i];
  atomic_fetch_add(sum[0], r);
  item.barrier(access::fence_space::local_space);
  if (lid == WGS-1) {
    int partial_sum = atomic_load(sum[0]);
    atomic_fetch_add(out[0], partial_sum);
  }
});
```

**Listing 4: The SYCL kernel scope of the hierarchical reduction with parameterized workload size**

## 4 Experiment

### 4.1 Setup

We evaluate the reductions on an Intel integrated GPU (UHD Graphics 630) and two Nvidia discrete GPUs (P100 and V100). The architecture of the integrated GPU is Coffee Lake GT2, Generation 9.5 [25]. It has 24 compute units running at 1.15 GHz. The maximum work-group size supported by the device driver is 256. The P100 has 56 multiprocessors. The default application clock speed for the graphics is 1.189 GHz and the memory clock 715 MHz. The V100 has 80 multiprocessors. The default application clock speed for the graphics is 1.312 GHz and the memory clock 877 MHz. The maximum work-group size supported by the device driver is 1024. For the BabelStream benchmark, the maximum memory bandwidths we observe are approximately 37 GB/s, 426 GB/s, and 853 GB/s on the UHD630, P100, and V100 GPUs, respectively. When targeting the Intel GPU, we build the SYCL programs with the Gold release of the Intel oneAPI Base Toolkit. When targeting the Nvidia GPUs, we build the same SYCL programs with the CUDA-enabled SYCL compiler. The versions of the CUDA software development kit are 11.0 and 11.2 for the P100 and V100 GPUs, respectively. The compilers' optimization option is "-O3".

We measure the kernel execution time of the sum reduction on a device for performance evaluation. While an integrated GPU does not incur PCIe communication overhead, it is not designed to match the raw performance of a discrete GPU [25]. Hence, our evaluation is focused on the impacts of reduction implementations upon the kernel performance for each GPU. The number of integers to reduce on a GPU are 1048576000, approximately 4 GB in memory size. Reduction over sufficiently large numbers may mitigate variances in observed kernel execution time across the range of work-group size. Kernel time is measured with the command-line profilers: the intercept layer for OpenCL applications [26] and the Nvidia performance profiler [27]. We use the average execution time of 100 invocations of a kernel for the reduction performance. The GPU results are verified by comparing them with the expected results.

### 4.2 Experimental Results

Figure 2 shows the execution time in milliseconds (ms) of the kernels shown in Listing 3 with respect to the work-group sizes and vector widths on the UHD630. The work-group size ranges from 16 to 256 and the vector width from 1 to the maximum value of 16. We omit kernel time for the work-group sizes ranging from 1 to 8 because their execution time becomes significantly longer. For a given vector width, the work-group sizes ranging from 64 to 256 see the lowest execution time. On the other hand, a vector width of four sees the lowest execution time for most work-group sizes. The execution time begins to increase when the width increases from 4 to 8 or from 8 to16. When the work-group size is 64 and the vector width is 4, the minimum execution time is 114.3 ms. When the work-group size is 16 and the vector width is 1, the longest execution time is 749.3 ms. Hence, tuning the work-group size and vector width can improve the performance by a factor of 6.6.

Figure 3 shows the execution time in milliseconds of the kernels shown in Listing 4 with respect to the work-group sizes and workload sizes on the UHD630. The workload size ranges from 1 to 32. Though the execution time decreases monotonically when the work-group size is 16 or 32, larger workload sizes increase the kernel execution time when the work-group sizes are over 32. We observe that the performance trend is similar to that shown in Figure 2. Hence, it is important to tune the two parameters for improving the performance. When the work-group size is 128 and the vector width is 4, the minimum kernel execution time is 110.9 ms, approximately 3% shorter than the minimum time shown in Figure 2. When the work-group size is 16 and the vector width is 1, the longest execution time is 758.5 ms. Hence, tuning the work-group size and vector width can speed up the performance by a factor of 6.8.

Figure 4 shows the execution time in milliseconds of the

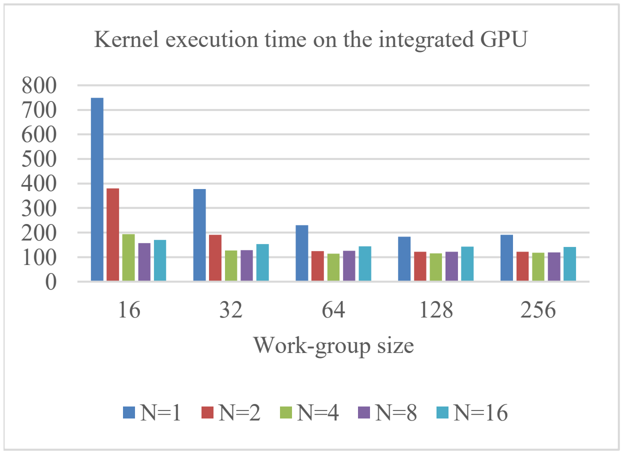

Figure 2: Average execution time of the kernels shown in Listing 3 with respect to the work-group sizes and vector widths (N) on the Intel UHD630 GPU

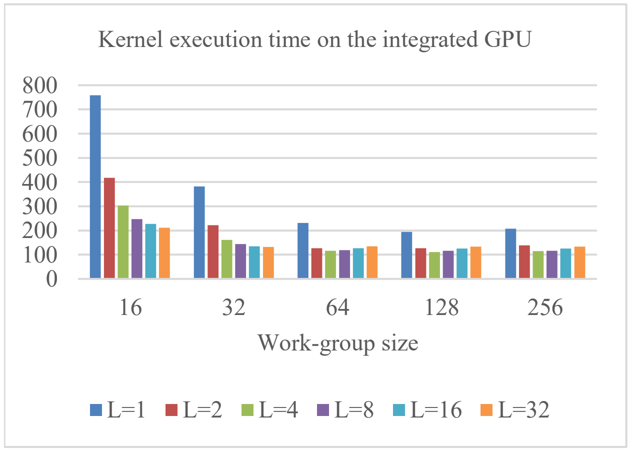

Figure 3: Average execution time of the kernels shown in Listing 4 with respect to the work-group sizes and workload sizes (L) on the Intel UHD630 GPU

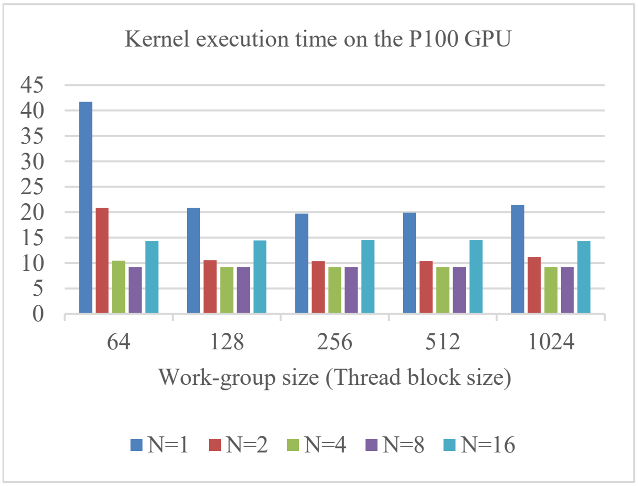

**Figure 4: Average execution time of the kernels shown in Listing 3 with respect to the work-group sizes and vector widths (N) on the Nvidia P100 GPU**

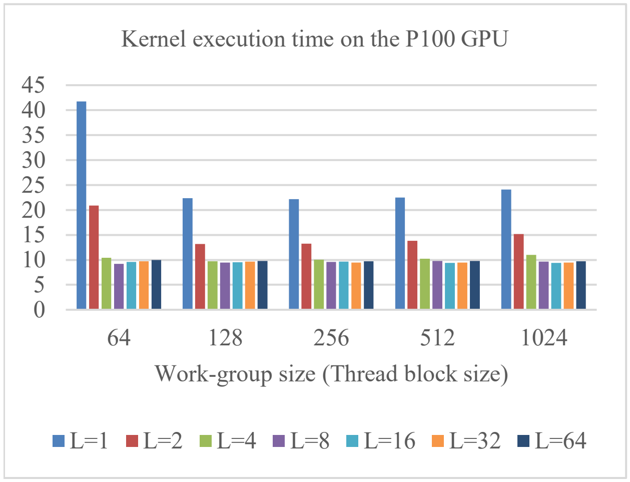

**Figure 5: Average execution time of the kernels shown in Listing 4 with respect to the work-group sizes and workload sizes (L) on the Nvidia P100 GPU**

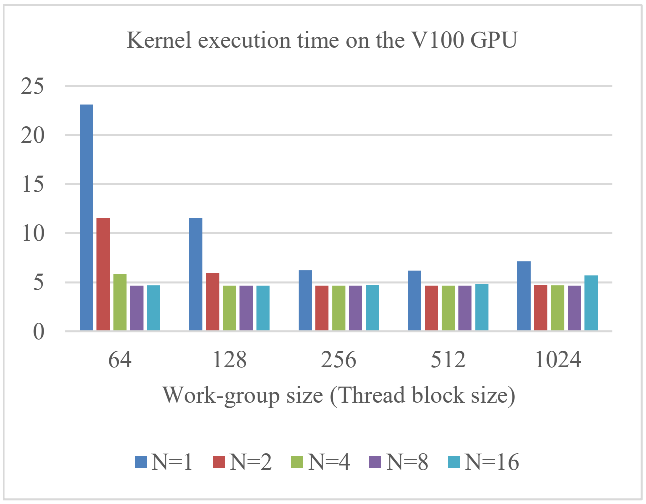

**Figure 6: Average execution time of the kernels shown in Listing 3 with respect to the work-group sizes and vector widths (N) on the Nvidia V100 GPU**

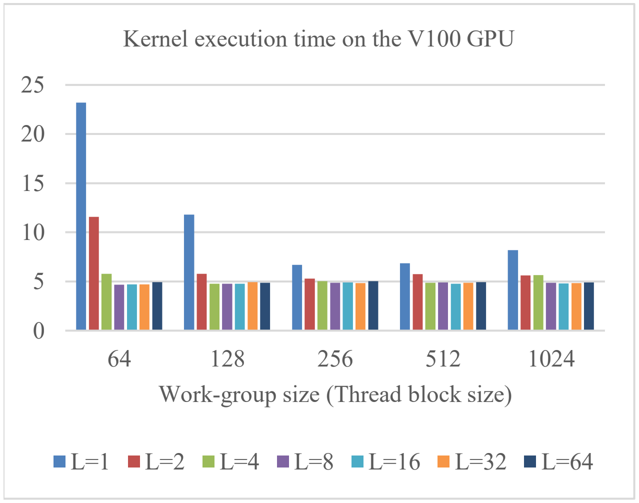

**Figure 7: Average execution time of the kernels shown in Listing 4 with respect to the work-group sizes and workload sizes (L) on the Nvidia V100 GPU**

kernels shown in Listing 3 with respect to the work-group sizes and vector widths on the P100. The work-group size ranges from 64 to 1024 so that the total number of work-group sizes is equal to the total number of work-group sizes selected for the Intel GPU. The range of the vector width is from 1 to 16. Interestingly, a vector width of four also sees the lowest execution time for most work-group sizes. On the other hand, a vector width of 16 increases the lowest kernel execution time across the work-group sizes by more than 50%. When the work-group size is 512 and the vector width is 4, the minimum execution time is 9.197 ms. When the work-group size is 64 and the vector width is 1, the longest execution time is 41.8 ms. Hence, tuning the work-group size and workload size can improve the performance by a factor of 4.6.

Figure 5 shows the execution time in milliseconds of the kernels shown in Listing 4 with respect to the work-group sizes and vector widths on the P100. The work-group size ranges from 64 to 1024. When the work-group size is 64 and the workload size is 8, the minimum execution time is 9.207 ms. When the work-group size is 64 and the workload size is 1, the longest execution time is 41.8 ms. Hence, tuning the work-group size and workload size can improve the performance by a factor of 4.5.

Figure 6 shows the execution time in milliseconds of the kernels shown in Listing 3 with respect to the work-group sizes and vector widths on the V100 GPU. When the work-group size is 128, 256 or 512, and the vector width is 4, the minimum execution time is 4.671 ms. When the work-group size is 64 and the vector width is 1, the longest execution time is 23.1 ms. Hence, tuning the work-group size and workload size can improve the performance by a factor of 4.9.

Figure 7 shows the execution time in milliseconds of the kernels shown in Listing 4 with respect to the work-group sizes and workload sizes on the V100 GPU. When the work-group size is 64, and the workload size is 16, the minimum execution time is 4.72 ms. When the work-group size is 64 and the vector width is 1, the longest execution time is 23.2 ms. Hence, tuning the work-group size and workload size can improve the performance by a factor of 4.9.

## 4.3 Performance Comparison with Open-source Implementations

We measure the performance of the sum reductions by evaluating open-source benchmarks and libraries. In the OpenCL reduction benchmark [28], three OpenCL kernels were developed using shared local memory, sub-group reduction, and work-group reduction, respectively. The latter two kernels require the OpenCL 2.0 support.

The SYCL compilers have been supporting a framework-agnostic reduction class (reducer) that hides the implementation details of reductions, allowing for the specification of a reduction operator using a function object. The code snippet in Listing 5 shows that the SYCL reducer performs the sum reduction by calling the "combine" method. The details of the reduction object and its usage are described in [29].

```
cgh.parallel_for<class reduce>(
  nd_range<1>(gws, lws), reducer, [=]
  (nd_item<1> item, auto &sum) {
     int gid = item.get_global_id(0);
     sum.combine(input[gid]);
});
```

**Listing 5: The sum reduction using the SYCL reducer class**

Thrust provides templated interfaces to algorithms and data structures designed for high-performance heterogeneous computing [30]. Thrust abstractions are agnostic of any parallel framework. The "thrust::reduce" function in Thrust, which is similar to the C++ Standard Template Library's "std::accumulate", computes the sum of all the elements under a specified range. The code snippet in Listing 6 demonstrates how "thrust::reduce" computes the sum of a sequence of integers through a device vector.

```
thrust::device_vector<int> d_input = input;
sum = thrust::reduce(d_input.begin(),
                      d_input.end(),
                      0, thrust::plus<int>());
```

**Listing 6: The sum reduction with Thrust**

CUB is a state-of-the-art library of collective primitives and utilities [31]. CUB is specific to CUDA C++ and its interfaces explicitly accommodate CUDA-specific features. CUB provides device-wide, parallel reductions across a sequence of data items residing within device-accessible memory [32]. The code snippet in Listing 7 shows how the reduction computes a sequence of "M" integers through a pre-allocated temporary storage on a GPU device.

```
cub::DeviceReduce::Sum(d_temp_storage,
  temp_storage_bytes, d_input, d_sum, M);
```

**Listing 7: The sum reduction with CUB**

Figure 8 show the execution time in milliseconds of six kernels on the UHD630 when the work-group size ranges from 16 to 256. "k1", "k2", and "k3" represent the three OpenCL kernels, respectively. "k4" constructs a SYCL reduction object for integer sum. For "k5" and "k6", we select the ones that achieve minimum execution time in Figure 2 and Figure 3, respectively. The minimum execution time among the first four kernels is 516 ms, 258.3 ms, 229.7 ms, 158 ms, and 182 ms for the five work-group sizes, respectively. Hence, we can obtain a performance speedup ranging from 1.4 (the work-group size of 128) to 3.2 (the work-group size of 16) over the publicly available implementations.

Because the drivers of the two Nvidia GPUs do not fully support OpenCL 2.0 features, we focus on the performance of the reductions using CUDA libraries and SYCL reducer. On the P100 GPU, the execution time of the reduction kernels implemented with Thrust, CUB, and the SYCL reducer is 9.487 ms, 9.224 ms, and 40.4 ms, respectively. Hence, the fastest SYCL implementation is approximately 3% and 0.3% faster than the templated reduction in Thrust and the device reduction in CUB, respectively. On the V100 GPU, the execution time of the reduction kernels implemented with Thrust, CUB, and the SYCL reducer is 4.76 ms, 4.69 ms, and 11.62 ms, respectively. Hence, the fastest SYCL implementation is approximately 1.9% and 0.4% faster than the templated reduction in Thrust and the device

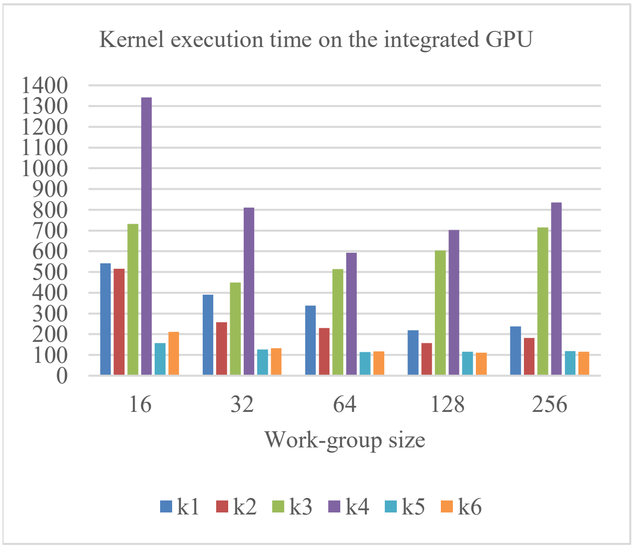

**Figure 8: Average execution time of the six kernels with respect to the work-group sizes on the Intel UHD630 GPU**

reduction in CUB, respectively.

The performance comparison indicates that Thrust and CUB are mature libraries for parallel reduction on an Nvidia GPU. However, there is a large optimization space for the SYCL reducer class.

## 5 Related Work

Previous studies characterized the performance of reductions and atomic functions on Nvidia GPUs [33, 34, 35]. In [31], the author evaluates the performance of seven reduction kernels over 4M numbers on an Nvidia G80 GPU. Each kernel improves the performance of the previous one. The kernel, which achieves the highest performance, has each thread sum up multiple elements in a shared local memory. While the block size is limited by the GPU to 512 threads, a block size of 128 can reach the highest kernel performance on the G80. In [32], the authors point out that tree-based algorithms are quite fast, but they suffer from too much synchronization because a barrier is needed for each loop iteration. The number of elements that a thread loads from global memory can be tuned to achieve a certain speed-up, but they did not pay attention to the tuning. In [33], the authors discover that, with appropriate atomic collision reduction techniques, the atomic implementation can outperform the non-atomics implementation, even for benchmarks known to have high-performance non-atomics GPU implementations. Atomics could greatly reduce coding complexity as thread-private object management and explicit thread-communication (for the shared data objects protected by atomic operations) are not necessary.

Power-of-two work-group sizes between 64 and 256 are recommended by the vendor to keep the utilization of an Intel GPU high [36]. This is consistent with our findings. While early study shows that a vector width of four is the preferred size on Intel and Nvidia GPUs [37], the performance of the reduction also depends on work-group sizes. When the work-group size is 16 on the Intel GPU and 64 on the Nvidia GPUs, a vector with eight elements achieves higher performance.

While atomics should be used with caution due to the synchronization overhead, scientific applications can also achieve higher performance by parallelizing sequential executions with atomics on emerging accelerators [38, 39]. The author of the OpenCL benchmark evaluates the reduction performance for a work-group size of 256 and 6291456 integers (24MB) on an Intel HD Graphics 530. We evaluate the performance impact of work-group sizes on sufficiently large numbers of integers. As far as we know, the performance of the reducer class has not been published on GPUs of different vendors.

## 6 Conclusion

SYCL is a promising programming model for heterogenous computing. We explain our SYCL implementations of the integer sum reduction using shared local memory, atomic operations, vectorized memory accesses, and parametrized workload sizes. With the maturing SYCL compilers, we evaluate the performance of the reductions on the Intel and Nvidia GPUs. The results show that tuning work-group sizes, vector widths, and workload sizes are important for performance improvement for integrated and discrete GPUs. Compared to the performance of the OpenCL kernels and SYCL reducer, our SYCL implementations can achieve 1.4X speedup on an Intel UHD630 GPU. On an Nvidia P100 GPU, our implementations are 3% and 0.3% faster than the templated reduction in Thrust and the device reduction in CUB, respectively. On an Nvidia V100 GPU, our implementations are 1.9% and 0.4% faster than the templated reduction in Thrust and the device reduction in CUB, respectively. Thrust and CUB are mature libraries for parallel reduction on an Nvidia GPU. However, there is a large optimization space for the SYCL reducer class. While the number of integers to reduce are not arbitrary in our implementations, the potential of performance improvement using the SYCL programming model will drive the portability path with the development of SYCL compilers. As future work, we will investigate the performance of SYCL applications that contain reduction kernels.

## ACKNOWLEDGMENTS

We appreciate the reviewers' criticism and comments. This research used resources of the Experimental Computing Laboratory at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. The research also used resources of the Intel DevCloud.

REFERENCES

[1] Munshi, A., Jacobs, I.S., Bean, C.P., Rado, G.T. and Suhl, H., 2007. Khronos OpenCL Working Group. The OpenCL Specification, Version 1, pp.271-350.

[2] Czajkowski, T.S., Aydonat, U., Denisenko, D., Freeman, J., Kinsner, M., Neto, D., Wong, J., Yiannacouras, P. and Singh, D.P., 2012, August. From OpenCL to high-performance hardware on FPGAs. In 22nd International Conference on Field Programmable Logic and Applications (pp. 531-534). IEEE.

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

[4] https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf

[5] Ke, Y., Agung, M. and Takizawa, H., 2021, January. neoSYCL: a SYCL implementation for SX-Aurora TSUBASA. The International Conference on High Performance Computing in Asia-Pacific Region (pp. 50-57).

[6] Deakin, T. and McIntosh-Smith, S., 2020, April. Evaluating the performance of HPC-style SYCL applications. In Proceedings of the International Workshop on OpenCL (pp. 1-11).

[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] Christgau, S. and Steinke, T., 2020, May. Porting a Legacy CUDA Stencil Code to oneAPI. In 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (pp. 359-367). IEEE

[9] Constantinescu, D.A., Navarro, A., Corbera, F., Fernández-Madrigal, J.A. and Asenjo, R., 2020. Efficiency and productivity for decision making on low-power heterogeneous CPU+ GPU SoCs. The Journal of Supercomputing, pp.1-22.

[10] Johnston, B., Vetter, J.S. and Milthorpe, J., 2020, November. Evaluating the Performance and Portability of Contemporary SYCL Implementations. In 2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC) (pp. 45-56). IEEE.

[11] Joó, B., Kurth, T., Clark, M.A., Kim, J., Trott, C.R., Ibanez, D., Sunderland, D. and Deslippe, J., 2019, November. Performance portability

of a Wilson Dslash stencil operator mini-app using Kokkos and SYCL. In 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (pp. 14-25). IEEE.

[12] Reguly, I.Z., 2019, November. Performance portability of multi-material kernels. In 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC) (pp. 26-35). IEEE.

[13] Jin, Z. and Finkel, H., 2019, November. Evaluation of Medical Imaging Applications using SYCL. In 2019 IEEE International Conference on Bioinformatics and Biomedicine (pp. 2259-2264). IEEE.

[14] Afzal, A., Schmitt, C., Alhaddad, S., Grynko, Y., Teich, J., Forstner, J. and Hannig, F., 2018, July. Solving Maxwell's Equations with Modern C++ and SYCL: A Case Study. In 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP) (pp. 1-8). IEEE.

[15] Da Silva, H.C., Pisani, F. and Borin, E., 2016, October. A comparative study of SYCL, OpenCL, and OpenMP. In 2016 International Symposium on Computer Architecture and High-Performance Computing Workshops (SBAC-PADW) (pp. 61-66). IEEE.

[16] 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.

[17] Harvey, M.J. and De Fabritiis, G., 2011. Swan: A tool for porting CUDA programs to OpenCL. Computer Physics Communications, 182(4), pp.1093-1099.

[18] Perkins, H., 2017, May. CUDA-on-CL: a compiler and runtime for running NVIDIA CUDA C++ 11 applications on OpenCL™ 1.2 Devices. In Proceedings of the 5th International Workshop on OpenCL (pp. 1-4).

[19] Sathre, Paul, Mark Gardner, and Wu-chun Feng. On the portability of CPU-accelerated applications via automated source-to-source translation. In Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, pp. 1-8. 2019

[20] Babej, M. and Jääskeläinen, P., 2020, April. HIPCL: Tool for Porting CUDA Applications to Advanced OpenCL Platforms Through HIP. In Proceedings of the International Workshop on OpenCL (pp. 1-3).

[21] 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.

[22] https://github.com/intel/llvm

[23] https://github.com/intel/llvm/blob/sycl/sycl/doc/CompilerAndRuntimeDesign.md

[24] NVIDIA, NVIDIA Compute PTX: Parallel Thread Execution, 1st ed., NVIDIA Corporation, Santa Clara, California, October 2008

[25] Gera, P., Kim, H., Kim, H., Hong, S., George, V. and Luk, C.K., 2018, April. Performance characterisation and simulation of intel's integrated GPU architecture. In 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (pp. 139-148). IEEE.

[26] Ashbaugh, B., 2018, May. Debugging and Analyzing Programs Using the Intercept Layer for OpenCL Applications. In Proceedings of the International Workshop on OpenCL (pp. 1-2)

[27] Bradley, T., 2012. GPU performance analysis and optimisation. NVIDIA Corporation.

[28] https://github.com/ekondis/cl2-reduce-bench

[29] https://github.com/intel/llvm/tree/sycl/sycl/doc/extensions/Reduction

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

[31] Merrill, D., 2015. CUB: A pattern of “collective” software design, abstraction, and reuse for kernel-level programming. Nvidia Research.

[32] https://nvlabs.github.io/cub/

[33] Mark, H., 2008. Optimizing parallel reduction in CUDA. NVIDIA CUDA SDK.

[34] Martín, P.J., Ayuso, L.F., Torres, R. and Gavilanes, A., 2012, July. Algorithmic strategies for optimizing the parallel reduction primitive in CUDA. In High Performance Computing and Simulation (HPCS), 2012 International Conference on (pp. 511-519). IEEE.

[35] Egielski, I.J., Huang, J. and Zhang, E.Z., 2015. Massive atomics for massive parallelism on GPUs. ACM SIGPLAN Notices, 49(11), pp.93-103

[36] OpenCL Developer Guide for Intel® Processor Graphics. 2019 Update 4.

[37] Thoman, P., Kofler, K., Studt, H., Thomson, J. and Fahringer, T., 2011, August. Automatic OpenCL device characterization: Guiding optimized kernel design. In European Conference on Parallel Processing (pp. 438-452). Springer, Berlin, Heidelberg.

[38] Ramanathan, N., Wickerson, J., Winterstein, F. and Constantinides, G.A., 2016, February. A case for work-stealing on FPGAs with OpenCL atomics. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (pp. 48-53). ACM

[39] Jin, Z. and Finkel, H., 2018, May. Nuclear Reactor Simulation on OpenCL FPGA: a Case Study of RSBench. In Proceedings of the International Workshop on OpenCL (pp. 1-9).