applied sciences

MDPI

Article

Large-Scale Data Computing Performance Comparisons on SYCL Heterogeneous Parallel Processing Layer Implementations

Woosuk Shin [1], Kwan-Hee Yoo [2] and Nakhoon Baek [1,*]

[1] School of Computer Science and Engineering, Kyungpook National University, Daegu 41566, Korea; w.shin@knu.ac.kr

[2] Department of Computer Science, Chungbuk National University, Cheongju 28644, Korea; khyoo@cbnu.ac.kr

* Correspondence: nbaek@knu.ac.kr

Received: 6 January 2020; Accepted: 20 February 2020; Published: 1 March 2020

Abstract: Today, many big data applications require massively parallel tasks to compute complicated mathematical operations. To perform parallel tasks, platforms like CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language) are widely used and developed to enhance the throughput of massively parallel tasks. There is also a need for high-level abstractions and platform-independence over those massively parallel computing platforms. Recently, Khronos group announced SYCL (C++ Single-source Heterogeneous Programming for OpenCL), a new cross-platform abstraction layer, to provide an efficient way for single-source heterogeneous computing, with C++-template-level abstractions. However, since there has been no official implementation of SYCL, we currently have several different implementations from various vendors. In this paper, we analyse the characteristics of those SYCL implementations. We also show performance measures of those SYCL implementations, especially for well-known massively parallel tasks. We show that each implementation has its own strength in computing different types of mathematical operations, along with different sizes of data. Our analysis is available for fundamental measurements of the abstract-level cost-effective use of massively parallel computations, especially for big-data applications.

Keywords: single-source DSL (Domain Specific Language); heterogeneous computing; parallel computing; GPGPU (General Purpose Graphics Processing Unit)

1. Introduction

Nowadays, many large-scale data applications in various fields require massively parallel operations to obtain results efficiently. For massively parallel computations, we can use well-known system-level programming platforms including CUDA (Compute Unified Device Architecture), OpenCL (Open Computing Language), and ROCm (Radeon Open Compute). Recently, programmers in the massively-parallel computation field found that the platform-independence and the high-level abstraction are also important, in addition to the cost-effectiveness [1–4].

This paper focuses on the new parallel computing implementations of SYCL (C++ Single-source Heterogeneous Programming for OpenCL) [5,6], which are actually C++ template-level implementations of high-level massively parallel computation libraries. Since SYCL can be implemented over the existing standards of OpenCL and/or CUDA, we can indirectly achieve the platform-independence. The programmers can interact with abstracted SYCL function calls, and SYCL will do complex jobs with CUDA and/or OpenCL. Additionally, C++ template features provide the high-level abstractions to the programmers.

Although the Khronos Group introduced this well-designed standard of SYCL [5,6], they only provide the official specifications, without reference implementations. These days, the SYCL

programmers can use various SYCL implementations from different vendors. Unfortunately, an efficient SYCL implementation requires deep understanding of C++ templates, CUDA, OpenCL, and the parallel computation itself, and current SYCL implementations show diverse performances.

We will represent how heterogeneous parallel computing was historically implemented before the SYCL. We will compare the characteristics of existing SYCL implementations among various vendors. Finally, we compare the performance of those SYCL implementations by measuring execution times for well-known massively parallel computation problems.

Since most of the big-data computation and the big-data visualization applications need typical operations provided by SYCL (and underlying OpenCL and/or CUDA), this comparison and analysis provides fundamental information on the abstract-level cost-effective implementations of massively parallel computing, especially for big-data applications. This work is the first on the performance of SYCL implementations to the best of our knowledge.

2. Previous Works

2.1. Parallel Computing Platforms

Efforts to implement parallel computing have been undertaken mainly by hardware architecture researchers. The instruction-level parallelism is an important method to utilize CPU resources efficiently. The task parallelism is also important to distribute tasks across different processing cores. The conventional hardware parallelism assumes hardware locality.

In the application level, threading methods are widely used to control concurrency of the program. One of the most well-known abstractions to control concurrency will be *OpenMP* (Open Multi-Processing) [7]. OpenMP is an API (application programming interface) that provides programmers a simple and flexible interface for developing parallel applications on heterogeneous platforms. Since version 4.5, OpenMP has also supported GPGPU (general purpose graphics processing unit) accelerators with combinations of device directives [7,8].

Although OpenMP provides an efficient method to off-load task to GPU, OpenMP lacks some functionality to fully utilize modern GPU architecture. Modern GPGPU utilizes the programmable shared memory which usually resides in the L1 cache of GPU. Shared memory is shared within a compute unit (CU) or stream multiprocessor (SM), a set of cores that can be launched at the same time. In the latest NVIDIA Turing architecture, a single stream multiprocessor consists of 64 cores [9].

Currently, OpenMP do not provide API to control the shared memory of GPUs. A keyword shared used in OpenMP provides a single variable that can be shared between threads to store results of reduction operations, whereas shared memory is utilized to store intermediate information of parallel tasks. Without shared memory, performances are dramatically decreased. In this paper, we focused on the platforms with shared memory control features, excluding OpenMP.

Another well-known programming standard for parallel computing is OpenACC (Open Accelerations) [10,11]. OpenACC also controls parallelism by using programming language directives. Currently, OpenACC-capable compilers are limited to OpenUH (Open Source University of Houston) [12], OpenARC (Open Accelerator Research Compiler) [13] and Omni Compiler [14]. Moreover, these OpenACC-supporting compilers are limited to specific operating systems and parallel processors.

Kokkos [15] is an implementation of abstracted parallel execution and memory model. Kokkos gives great flexibility by complying with standard C++ and can be compiled using major compilers, including GNU C++ compiler and Clang compiler. Kokkos utilizes OpenMP, Pthreads, and CUDA to launch parallel tasks for heterogeneous devices. However, Kokkos lacks OpenCL support, and it limits its use for GPU-based parallel processors.

2.2. CUDA (Compute Unified Device Architecture), OpenCL (Open Computing Language) and SYCL (C++ Single-Source Heterogeneous Programming for OpenCL)

With the emergence of applications requiring massively parallel operations, they developed a set of parallel programming platforms (or frameworks) like CUDA, OpenCL and ROCm. Such parallel processor based platforms are designed for sequential parallelism, without requiring concurrency of threads. The architectural differences between two main platforms, CUDA and OpenCL, is well described in Figure 1 [1]. These platforms utilize GPUs as parallel processors. When GPUs are used for general-purpose data processing, rather than the traditional purpose of computer graphics processing, we call this GPGPU (general purpose GPU). The GPU processing is based on the *SPMD* (single program multiple data) paradigm, which is originated from graphics data processing.

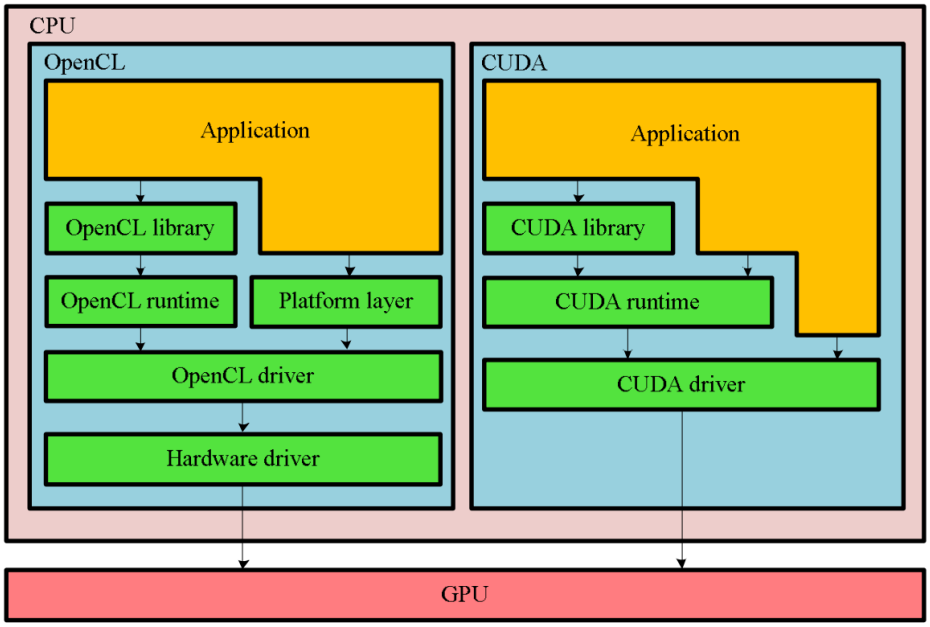

Figure 1. OpenCL (Open Computing Language) and CUDA (Compute Unified Device Architecture) framework [1].

Many parallel programming applications require massively parallel tasks to compute complicated mathematical operations. Matrix multiplications (including filtering) and vector additions (or matrix additions) are the most important among those most demanding and resource-consuming operations. These two operations are widely used, from the traditional image-processing applications to the new deep learning applications and typical big-data analysis applications.

The major differences between parallel programming platforms and traditional parallel processors, including multi-core CPUs, specially-designed FPGA (field-programmable gate array) and ASIC (application-specific integrated circuit) are modern parallel programming platforms of CUDA, and OpenCL allows us to write the parallel programming codes in high-level languages to give different instructions, not depending on the underlying chipsets.

These platforms are based on the SPMD paradigms, and process multiple data with the same instructions with the kernels. A *kernel* is a set of data and instructions that can be written in high-level languages (or assembly languages) for specific chipsets. These kernels are converted into machine-level instructions of parallel processing units, by native compilers like NVCC (NVIDIA CUDA Compiler) or HCC (Heterogeneous Compute Compiler). Such kernels can be also adapted to use LLVM (low-level virtual machine) concepts, as a good example, with which OpenCL supports various types of parallel processing units. Each kernel is guaranteed to run in a single thread to provide data consistency. Data used in each kernel can be provided by the host (processor), which also controls kernel launches.

One of the important differences between CUDA and OpenCL is that the former is NVIDIA GPU-dependent architecture, whereas the latter pursues hardware independency. Additionally, OpenCL uses queueing systems to minimize CPU overheads. The internal execution model with a single OpenCL program is well described in [16].

Even though OpenCL resolved hardware dependency on the existing parallel computing platforms, it also has some limitations. First, OpenCL kernels should be declared outside the main programming codes (separately as a character string or as a disk file), and then they should be compiled and linked during run-time of the main program. Although this run-time compiling of the kernel code gives a huge flexibility to programmers, embedding high-end Just-In-Time compiler and linker into the program requires many computational resources. Therefore, OpenCL run-time compiler omitted a few optimization tools to minimize required computational resources, leading to optimization being performed manually by programmers [17].

To minimize efforts of complicated programming processes and to support hardware-independent heterogeneous parallel computing ability, Khronos group announced SYCL. Figure 2 shows a framework diagram of SYCL for OpenCL. As described in the SYCL specification [4], SYCL is a cross-platform abstraction C++ programming model for OpenCL. SYCL uses a SMCP (single source multiple compiler passes) approach to compile their codes to generate a SYCL file, which can be executed by both CPU and parallel processors.

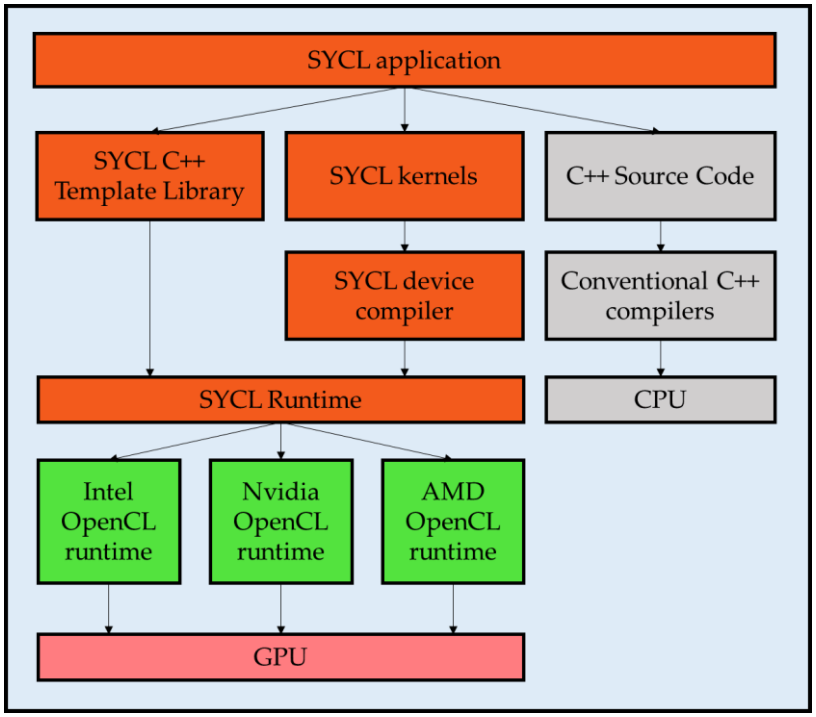

Figure 2. SYCL (C++ Single-source Heterogeneous Programming for OpenCL) for OpenCL framework.

Using OpenCL as a back-end inherits hardware independency. If a new chipset vendor provides its new OpenCL driver, we can naturally execute our SYCL codes on that chipset. The major differences between OpenCL and SYCL is that SYCL pursues minimal usage of device control functions, such as acquiring device information, creating context, creating a command queue, allocating memory, and others. In SYCL, these device control functions are not explicitly controlled by application programs, and are executed implicitly.

There are comparative studies to compare the efficiency of parallel-computation platforms [18–21]. According to [18], SYCL takes longer time to execute common midpoint operation, compared to OpenCL and OpenMP. However, in the point of memory usages, SYCL used the least memory amount

among another platforms. Additionally, kernel source code sizes are the shortest, showing better programmability than other parallel processing platforms. In [20], another well-known parallel processing platform kokkos, QPhiX (QCD for Intel Xeon Phi and Xeon processors), and QUDA (QCD on CUDA) is demonstrated and performs better than SYCL, showing greater GFLOPS (Giga Floating point Operations Per Second) on Lattice QCD (quantum chromodynamics) operations. For more complicated operations, deep neural network implementation for SYCL (SYCL-DNN) was compared to Intel math kernel library for deep neural networks in [21].

2.3. Translations between CUDA and OpenCL

As the parallel programming platform of CUDA was announced, utilizing GPU as a parallel processor has become easier. Many programs are written in CUDA to utilize GPGPU functionalities. However, since CUDA has been exclusively developed by NVIDIA, it supports only NVIDIA chip-sets. On the other hand, OpenCL, which was announced by Khronos group, is an efficient way to support various types of chipsets. In some cases, we can convert CUDA source codes to OpenCL source codes, to support wider ranges of parallel processors.

For the translation procedure between CUDA and OpenCL, we need to check the details of each parallel processing platform to launch their own kernels. Each parallel programming framework provides API bound to C/C++. They have distinguishable function names to launch kernel functions to control the execution flow, or to manipulate kernel data. For example, CUDA has *cuda* prefix to their API, OpenCL has *cl* prefix, and ROCm has *hip* prefix. Although each platform provides different function names, the processes of using such API in a system are very similar to each other.

First, the procedure will allocate resources in the parallel processor memory space. Second, the system will provide a kernel (which can be pre-compiled) that will be executed by each thread in parallel processors. Then, after launching the kernel, the system waits for all threads to finish their own works. Finally, the system copies back the final data, which contains results of thread execution.

A most intuitive way of the code translation is to let programmers manually translate the code of each platform by substituting similar API functions. There are also many efforts to achieve automatic translations, as shown in [22–24]. Researches show that the automatically translated code also performs well, at least for the case of converting CUDA codes to OpenCL codes.

However, translating codes written for one platform to another platform is still challenging. This is mainly due to the differences in the work allocation models, memory space architecture, and synchronization techniques. Although the single source heterogeneous computing concept allows compilers to generate compiler intermediate codes, generating another full source code for another platform has not yet been considered. To follow the concepts of single source heterogeneous parallel computing, SYCL was announced by Khronos group. We will show the details of SYCL in the next section.

3. Comparisons of SYCL Implementations

Recently, Khronos group released SYCL version 2.2 [6], which is based on the OpenCL 2.0 runtime functionalities. There is also another widely-used release of SYCL 1.2 [5], based on the implementation of OpenCL 1.2 run-time. As a consequence, SYCL 1.2 and SYCL 2.2 are mixed in the current SYCL market. From the point of heterogeneous computing view, selecting a more widely used version of SYCL is important to support various types of parallel processors.

Table 1 shows supported OpenCL versions for major chipsets. As shown in this table, most modern GPUs support OpenCL version 2.0 or higher. However, some mobile processors and all NVIDIA GPUs do not support the latest OpenCL versions. Therefore, this has difficulty in measuring performance differences between different platforms. For this reason, most SYCL implementations still support only SYCL 1.2 specification. Thus, in this paper, we focused on SYCL version 1.2.

Table 1. Supported OpenCL versions for major chipsets.

Vendor	Chipset	OpenCL Version
AMD	AMD GCN (Graphics Core Next) GPU	2.0+
ARM	Mali (bifrost)	2.0+
Intel	Intel 5th,6th gen processors	2.0+
Intel	Xeon Phi Processors	1.2
NVIDIA	GPU later than Kelper	1.2
Qualcomm	Adreno 500 series	2.0+

The purpose of SYCL is to provide more efficient abstract layers for parallel tasks. Figure 3 shows a typical example of the vector addition program written in SYCL. As SYCL aims for minimum device control function calls, manually copying data to host memory or device memory is not required before and after kernel launches. Another big difference from the traditional parallel processing platform is that kernel itself should be declared inside of the lambda capture. This is done by embedding the kernel code inside the C++ code, which results in SYCL itself being compiled by conventional C++ compilers. SYCL also allows the declaration of lambda capture inside kernels. Using nested lambda capture, SYCL also provides efficient and easy method to utilize nested parallelism. Widely used GPGPU parallelism platforms do not support nested parallelism. The concept and benefit of nested parallelism is well explained in [25]. According to Khronos group, at this time, there are three well-known SYCL implementations: ComputeCpp, triSYCL, and hipSYCL

```
main() {
  std::vector h_a(SIZE), h_b(SIZE), h_c(SIZE); //host vectors
  ...
  initialize host vector's data
  ...
  {
    Buffer d_a(h_a); //automatically copies data to device
    Buffer d_b(h_b); //also, allocates device memory
    Buffer d_c(h_c); //with the same size of host memory

    queue q;
    command_group(q, [&]() {
    auto a = d_a.get_access<access::read>();
    auto b = d_b.get_access<access::read>();
    auto c = d_c.get_access<access::write>();

    parallel_for(SIZE, vector_addition([=](id<> item) {
        int i = item.get_global(0);
        c[i] = a[i] + b[i];
      }
    }
  }
}
```

Figure 3. A vector addition program written in SYCL.

The ComputeCpp is a heterogeneous parallel programming platform developed by Codeplay Inc, Scotland. ComputeCpp provides a conformant implementation of SYCL 1.2.1 Khronos specification. ComputeCpp provides the parallel processing capability through generating intermediate codes based on SPIR (Standard Portable Intermediate Representation) or SPIR-V that targets OpenCL. It also can generate NVPTX (NVIDIA Parallel Thread Execution) source codes, which target NVIDIA GPU.

The triSYCL is an open source implementation with the specification of the SYCL C++ layer. The triSYCL implementation fully utilizes functionalities provided by the Boost.Compute library [26].

However, although Boost.Compute library supports OpenCL kernel codes to be compiled and executed, triSYCL does not utilize such functionality. The major drawback of not using OpenCL functionality is that the triSYCL does not support local memory. Local memory (or equivalently, shared memory in CUDA) is important storage to reduce gaps between global memory and registers. As a result, some complex operations like matrix-multiplication are not ideal to be executed on the triSYCL implementations.

The hipSYCL is built to utilize NVIDIA CUDA and AMD HIP (C++ Heterogeneous-Compute Interface for Portability). That is, with a single source code, hipSYCL can generate hardware specific executable binary codes for CUDA and/or ROCm, rather than OpenCL. Although it is in its early stage of development, one huge advantage of generating CUDA or HIP native binary source code is that there are many optimizations which OpenCL does not comply with. For example, although the queue concept of OpenCL is implemented to improve its performance by releasing the CPU workloads, it is faster to launch a kernel in CUDA if the kernel has only few operations.

There are studies about the performance comparison between platforms, as shown in [18], or experiments using triSYCL, as shown in [27]. However, there are very few studies about performance measures on each SYCL implementation.

4. Experimental Results

In our experiment, we measure execution times for the most widely used parallel operations of vector additions and matrix multiplications on each SYCL implementation. The execution includes data copies, kernel launch overheads, and operations in the kernels. We also measure total execution times of the same operations performed on CUDA platforms, for comparison purposes. By measuring those CUDA execution times, we can analyze gaps between native parallel programming platforms and SYCL abstracted parallel programming layers.

For all experiments, all data are generated randomly at the initial stage of the main program, before launching the parallel tasks. Considering that vectors and matrices can be represented in a one-dimensional array, data given to parallel processor are two input data arrays and single result array. The same type and size of data was given to launch parallel tasks between CUDA and SYCL. The value of individual elements inside each array is a floating-point number between zero and one. After parallel tasks are finished, we also performed same operations with CPU to check data consistency. The resulting output data of parallel tasks should have same value as the result of the CPU. Time to check consistency between results were excluded from execution time. The experiments were conducted with Intel i5-6500 CPU, Intel integrated Gen9 GPU, and NVIDIA GTX960 GPU built on Maxwell micro architecture.

Table 2 shows execution times of vector additions on various implementations with regards to the size (dimension) of vectors. It is clear that the execution time of the native platform CUDA is shorter than that of SYCL platforms for all vector sizes. Until vector size reaches 4096, triSYCL performs better than hipSYCL. With vector size at about 1 million, the execution time of hipSYCL is the fastest. However, with vector size over 10 million, ComputeCpp performs better than the other implementations.

Table 2. Execution time of vector addition.

Size	CUDA	HipSYCL	ComputeCpp	TriSYCL
32	45 µs	1900 µs	2960 µs	474 µs
1024	50 µs	2000 µs	3048 µs	580 µs
4096	71 µs	2000 µs	3044 µs	800 µs
1048576	6295 µs	2000 µs	31,513 µs	76,360 µs
16777216	100,119 µs	280,000 µs	117,690 µs	1,190,000 µs

Table 3 is the execution times of matrix multiplications, using only global memory. For the matrix multiplication, hipSYCL performs better until matrix size reaches 256. However, if matrix size is larger than 256, ComputeCpp performs better than other implementations. However, triSYCL implementation cannot calculate results within an adequate range of time for matrix size bigger than 1024. The difference between CUDA and hipSYCL occurs because of a different strategy in slicing threads into work-groups. Matrix multiplications with global memory require atomic operations to avoid memory conflict situations while writing data to memory. During atomic operations of a thread, another thread cannot access the memory and waits until the atomic operation ends. Thus, different work-group sizes, which launches concurrently, can affect the time to wait for atomic operations. This fact shows that managing work-group size is another important factor that should be optimized.

Table 3. Execution time of global memory matrix multiplication.

Size	CUDA	HipSYCL	ComputeCpp	TriSYCL
32	0.1 ms	2 ms	3 ms	5 ms
256	0.5 ms	14 ms	18 ms	1437 ms
1024	19 ms	730 ms	80 ms	-
2048	149 ms	5634 ms	584 ms	-

In Table 4, we also measure time to execute matrix multiplication using local memory. Since triSYCL does not support operations within local memory, the result is excluded from the table. For the matrix multiplication with local memory, ComputeCpp performs better than hipSYCL for matrix size bigger than 32. Also, the execution time of hipSYCL took longer than the matrix multiplication using global memory. From the data, we can see that context changing of local memory is occurring frequently in hipSYCL. Considering the reason for frequent local memory context swapping, we can estimate that the workgroup size of hipSYCL is set smaller compared to ComputeCpp. From the data, we can assume that hipSYCL performs well for limited data where ComputeCpp performs well for bigger data when local memory is used.

Table 4. Execution time of local memory matrix multiplication.

Size	CUDA	HipSYCL	ComputeCpp
32	0.1 ms	2 ms	3 ms
256	0.4 ms	25 ms	15 ms
1024	7.6 ms	1100 ms	59 ms
2048	60 ms	8765 ms	285 ms

An experimental result for differences between chipsets is shown in Table 5. The experiment is conducted using Intel Generation 9 based integrated GPU and NVIDIA GTX960 based on Maxwell architecture. When compiling the SYCL program, we utilized ComputeCpp SYCL implementation, since ComputeCpp supports the largest number of platforms. The result show that Intel GPU has limited computing power compared to NVIDIA chipsets. For operations not using local memory, operations with small data sizes took up to 46 times longer than NVIDIA chipset. Operations with large size of data take at most 2 times longer than NVIDIA chipset.

For Intel GPU, the execution time of matrix multiplication with local memory takes longer than operations without using local memory. The tendency of performance reduction is a result of two factors. The first reason is the architecture of chipset, as Intel chipset has fewer physical threads (execution unit) that can be launched simultaneously. As the number of execution units are small, multiple context switching is inevitable to execute large logical work-group items, which shares local memory. Another reason for decreased performance is the absence of dedicated GPU memory. In this

case, shared video memory should be utilized to store data. Since shared video memory should be shared with CPU, the data cannot be fetched as fast as dedicated memory, which leads to a slowdown of overall performance.

Table 5. Execution time of operations on NVIDIA and Intel graphics processing unit (GPU).

Operation	Size	Intel Generation 9	NVIDIA Maxwell
Vector Addition	32	88 ms	2.960 ms
	1024	88 ms	3.048 ms
	4096	88 ms	3.044 ms
	1,048,576	100 ms	31.513 ms
	16,777,216	257 ms	117.690 ms
Matrix Multiplication	32	140 ms	3 ms
	256	143 ms	18 ms
	1024	252 ms	80 ms
	2048	1148 ms	584 ms
Matrix Multiplication (Local Memory)	32	165 ms	3 ms
	256	170 ms	15 ms
	1024	382 ms	59 ms
	2048	1995 ms	285 ms

During experiments, we collected the profiled data for API calls of detailed execution of CUDA and hipSYCL with the NVIDIA profiler. The NVIDIA profiler provided fine information about runtime of CUDA, however, it did not provide a capability to profile OpenCL applications. On the other hand, CodeXL developed by AMD provided an OpenCL profiler, but profiling technology was limited to AMD GPU. Therefore, since the majority of our experiments were conducted with NVIDIA GPU, we could profile only two implementations run on CUDA. For profiling, we conducted matrix multiplications with the size of 1024, using shared memory.

Memory copy operation for hipSYCL was generated to utilize the asynchronous memory copy functionality of CUDA. Asynchronous memory copy allows programs to run another code during memory copy operations. However, according to Table 6, hipSYCL which utilized the asynchronous copy took longer than the synchronous copy. We can assume that when we execute only a single parallel task, asynchronous copy causes more overhead than sequential copy. However, when we have to launch multiple parallel task for large data, the synchronous copy will cause more overhead for each launch. Additionally, by comparing Table 4; Table 6, we can see that total execution time of a parallel task increased when we enabled profiling functions. That is, to achieve the best performance, profile functions must be disabled.

Table 6. Execution time of individual CUDA application programming interface (API).

API	CUDA	HipSYCL
cudaMemcpy (Host to Device)	319 µs	504 µs
Kernel Launch	8.18 ms	1.14 s
cudaMemcpy (Device to Host)	338 µs	537 µs

Table 7 shows detailed parameters during launch of the kernel and parallel computation. The result of Tables 6 and 7 show the importance of optimization in parallel task launch dimensions. SYCL implementations do not provide directives to control parallel task launch dimensions. Therefore, the launch dimensions are automatically selected by the implementation. However, as modern NVIDIA GPU can launch up to 1024 threads simultaneously, the dimensions smaller than 1024 used by hipSYCL cause frequent context changes, leading to lower theoretical occupancy.

Table 7. Detailed profiling information of each kernel launch.

Parameters	CUDA	HipSYCL
Grid Size (local work-group)	(32, 32, 1)	(64, 64, 1)
Block Size (global work-group)	(32, 32, 1)	(16, 16, 1)
Registers per Thread	24	42
Shared Memory used per block	8 KiB	2 KiB
Theoretical Occupancy	100%	63%
Shared Memory Executed	16 KiB	10 KiB

Moreover, older architecture like NVIDIA Fermi has only 32 registers available per thread. Therefore, a hipSYCL generated kernel code may lead to hazardous security issues during compilation or execution on GPUs with outdated architecture.

5. Conclusions

In this paper, we compared differences between SYCL implementations. Although SYCL specification itself aims for single source heterogeneous computing, there are some implementations which do not support full specification of SYCL. Each platform had its characteristics in managing data and work-group sizes, even though for the same task. To achieve better results, selecting the most suitable SYCL implementation is inevitable to maximize kernel performance.

For comparison purposes, we also measured the execution time of operations written in CUDA. The result shows that in most cases, CUDA showed better performance. This was mainly due to the fact that the native platform of CUDA provides the best optimization methods for their chipsets. From the view point of single source heterogeneous computing, SYCL is one of the best solutions to give flexibility in writing parallel tasks. SYCL also provides convenient nested parallelism by fully supporting native C++ standard lambda captures inside the kernel. Moreover, the SYCL implementation work groups are now focusing on optimizing their current implementations. Thus, we can expect performance enhancement as the SYCL implementations are optimized.

Conclusively, SYCL implementations are still slower than the original parallel programming platforms, but the applications are rapidly moving to the new flexible solution of SYCL. This trend is especially important for large-scale parallel computing applications, i.e., the big-data computing and the big-data visualization applications. Our comparisons will be the starting point for selecting the most suitable SYCL implementation for a specific task.

Author Contributions: Conceptualization, W.S., N.B.; formal analysis, W.S.; investigation, W.S.; writing—original draft preparation, W.S.; writing—review and editing, K.-H.Y. and N.B.; supervision, N.B.; Funding Acquisition, K.-H.Y. and N.B. All authors have read and agreed to the published version of the manuscript.

Funding: This work has supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (Grand No.NRF-2019R1I1A3A01061310). This research was supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2018-2015-0-00448) supervised by the IITP (Institute for Information and communications Technology Promotion). This research was also supported by the BK21 Plus project (SW Human Resource Development Program for Supporting Smart Life) funded by the Ministry of Education, School of Computer Science and Engineering, Kyungpook National University, Korea (21A20131600005).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Su, C.; Chen, P.; Lan, C.; Huang, L.; Wu, K. Overview and Comparison of OpenCL and CUDA Technology for GPGPU. In Proceedings of the 2012 IEEE Asia Pacific Conference on Circuits and Systems, Kaohsiung, Taiwan, 2–5 December 2012; pp. 448–451. [CrossRef]

2. Baek, N.; Yoo, K. Massively Parallel Acceleration Methods for Image Handling Operations. *Cluster Comput.* **2017**, *20*, 1149–1154. [CrossRef]
3. Baek, N.; Kim, J.K. An Artifact Detection Scheme with CUDA-based Image Operations. *Cluster Comput.* **2017**, *20*, 749–755. [CrossRef]
4. Baek, N. Design and Implementation of OpenGL SC 2.0 Rendering Pipeline. *Cluster Comput.* **2019**, *22*, 931–936. [CrossRef]
5. Keryell, R.; Howes, L. *SYCL 1.2.1 Specification Revision 5*; Khronos OpenCL Working Group—SYCL Subgroup: Beaverton, OR, USA, 2019.
6. Khronos OpenCL Working Group. *SYCL Provisional Specification, Version 2.2*; Khronos Group: Beaverton, OR, USA, 2016.
7. Antao, S.F.; Bataev, A.; Jacob, A.C.; Bercea, G.T.; Eichenberger, A.E.; Rokos, G.; Martineau, M.; Jin, T.; Ozen, G.; Sura, Z.; et al. Offloading Support for OpenMP in Clang and LLVM. In Proceedings of the 2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), Salt Lake City, UT, USA, 14 November 2016; pp. 1–11.
8. Liao, C.; Yan, Y.; de Supinski, B.R.; Quinlan, D.J.; Chapman, B. Early Experiences with the OpenMP Accelerator Model. *Lect. Notes Comput. Sci.* **2013**, *8122*, 84–98.
9. NVIDIA. *NVIDIA Turing Architecture Whitepaper*; NVIDIA Corporation: Santa Clara, CA, USA, 2018.
10. Wienke, S.; Springer, P.; Terboven, C.; an Mey, D. OpenACC—First Experiences with Real-World Applications. In *Proceedings of the European Conference on Parallel Processing*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 859–870.
11. Chandrasekaran, S.; Juckeland, G. *OpenACC for Programmers: Concepts and Strategies*; Addison-Wesley Professional: Boston, MA, USA, 2017.
12. Liao, C.; Hernandez, O.; Chapman, B.; Chen, W.; Zheng, W. OpenUH: An optimizing, portable OpenMP compiler. *Concurr. Comput. Pract. Experience.* **2007**, *19*, 2317–2332. [CrossRef]
13. Lee, S.; Vetter, J.S. OpenARC: Open Accelerator Research Compiler for Directive-Based, Efficient Heterogeneous Computing. In Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing, Vancouver, BC, Canada, 23–27 June 2014; pp. 115–120. [CrossRef]
14. Tabuchi, A.; Nakao, M.; Satao, M. A Source-to-Source OpenACC Compiler for CUDA. In *Proceeding of the European Conference on Parallel Processing*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 178–187.
15. Edwards, H.; Trott, C.R.; Sunderland, D. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel Distrib. Comput.* **2014**, *74*, 3202–3216. [CrossRef]
16. Munshi, A.; Gaster, B.; Mattson, T.G.; Ginsburg, D. *OpenCL Programming Guide*; Pearson Education: England, UK, 2011.
17. Komatsu, K.; Sato, K.; Arai, Y.; Koyama, K.; Takizawa, H.; Kobayashi, H. Evaluating Performance and Portability of OpenCL Programs. In Proceedings of the Fifth International Workshop on Automatic Performance Tuning, Berkeley, CA, USA, 22 June 2010; Volume 66, p. 1.
18. Silva, H.D.C.; Pisani, F.; Borin, E. A Comparative Study of SYCL, OpenCL, and OpenMP. In Proceedings of the 2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), Los Angeles, CA, USA, 26–28 October 2016; pp. 61–66. [CrossRef]
19. Joó, B.; Kurth, T.; Clark, M.A.; Kim, J.; Trott, C.R.; Ibanez, D.; Sunderland, D.; Deslippe, J. Performance Portability of a Wilson Dslash Stencil Operator Mini-App Using Kokkos and SYCL. In Proceedings of the2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), Denver, CO, USA, 22–22 November 2019; pp. 14–25.
20. Hammond, J.R.; Kinsner, M.; Brodman, J. A Comparative Analysis of Kokkos and SYCL as Heterogeneous, Parallel Programming Models for C++ Applications. In Proceedings of the International Workshop on OpenCL (IWOCL'19), Boston, MA, USA, 13–15 May 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 1–2. [CrossRef]
21. Burns, R.; Lawson, J.; McBain, D.; Soutar, D. Accelerated Neural Networks on OpenCL Devices Using SYCL-DNN. In Proceedings of the International Workshop on OpenCL (IWOCL'19), Boston, MA, USA, 13–15 May 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 1–4. [CrossRef]
22. Martinez, G.; Gardner, M.; Feng, W. CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures. In Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems, Taiwan, Taiwan, 7–9 December 2011; pp. 300–307. [CrossRef]

23. Gardner, M.; Sathre, P.; Feng, W.C.; Martinez, G. Characterizing the Challenges and Evaluating the Efficacy of a CUDA-to-OpenCL Translator. *Parallel Comput.* **2013**, *39*, 769–786. [CrossRef]
24. Kim, J.; Dao, T.T.; Jung, J.; Joo, J.; Lee, J. Bridging OpenCL and CUDA: A Comparative Analysis and Translation. In Proceedings of the SC '15 International Conference for High Performance Computing, Networking, Storage and Analysis, Austin, TX, USA, 15–20 November 2015; ACM: New York, NY, USA, 2015; pp. 82:1–82:12. [CrossRef]
25. Tanaka, Y.; Taura, K.; Sato, H.; Yonezawa, A. Performance Evaluation of OpenMP Applications with Nested Parallelism. In Proceedings of the International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers. LCR 2000. Lecture Notes in Computer Science, Rochester, NY, USA, 25–27 May 2000; Springer: Berlin/Heidelberg, Germany, 2002; Volume 1915.
26. Jakub, S. Boost.Compute: A Parallel Computing Library for C++ Based on OpenCL. In Proceedings of the 4th International Workshop on OpenCL (IWOCL '16), Vienna, Austria, 19–21 April 2016; Association for Computing Machinery: New York, NY, USA, 2016; pp. 15:1–15:39. [CrossRef]
27. Keryell, R.; Yu, L.Y. Early Experiments Using SYCL Single-Source Modern C++ on Xilinx FPGA: Extended Abstract of Technical Presentation. In Proceedings of the International Workshop on OpenCL (IWOCL '18), Oxford, UK, 14–16 May 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 18:1–18:8. [CrossRef]