

Development Tools for Heterogeneous Computing

Victor Smirnov(✉) and Nikita Storublevtcev(✉)

St. Petersburg State University, 7/9 Universitetskaya Nab,
St. Petersburg 199034, Russia
ariox41@gmail.com, 100.rub@mail.ru

Abstract. The problem of porting various applications to parallel computing accelerators, such as GPU and FPGA, is becoming ever more relevant. Such accelerators can significantly improve the performance and energy efficiency of applications.

In this paper, modern development tools focused on heterogeneous computing are reviewed. Particular attention is paid to the opportunities they provide for the porting of applications to different platforms with various architectures, and issues of application performance and reliability.

In particular, this article describes SYCL, a new standard for heterogeneous programming for C++, and presents the test results of one of its implementations -ComputeCpp.

ComputeCpp test results on the NVIDIA video card demonstrate the correctness of its implementation, but ComputeCpp performance was noticeably lower than CUDA. Given that ComputeCpp is in a beta state, there is hope that its performance will increase in the future, which will allow SYCL to become a real competitor to OpenCL in the development of portable applications for parallel accelerators.

Keywords: Parallel computing · GPGPU · Heterogeneous computing

1 Introduction

Currently, parallel data processing accelerators such as GPUs and FPGAs are increasingly being used to solve various computational problems. In many cases, they can improve the performance and energy efficiency of applications, and in some areas they allow to use of previously difficult-to-implement approaches, such as machine learning on mobile platforms. In this regard, the need to develop new applications for accelerators and to transfer old one's increases.

Developing an application for such accelerators requires the use of specialized and often new algorithms and development tools. The need to use specialized algorithms can be considered an inevitable evil, but questions arise about development tools. Although there is already a fairly stable ecosystem for heterogeneous computing: CUDA for NVIDIA graphics cards, HCC for AMD and OpenCL as a portable solution, portability between different platforms is still a problem. The quality of OpenCL

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11622, pp. 812–824, 2019.
https://doi.org/10.1007/978-3-030-24305-0_62

support varies for different platforms, but generally leaves much to be desired, and other solutions are non-portable.

In addition, there are more and more devices specializes for parallel data processing. Many processors contain embedded graphics cores (this is especially true for mobile platforms), there are processors with a large number of discrete cores that can also be effectively used for parallel data processing, and new GPUs and FPGAs are being developed.

Thus, the problem of choosing development tools suitable for specific computing platforms is becoming ever more relevant, and the need for portable application development tools that allow for truly heterogeneous computing is increasing.

2 Development Tools for Heterogeneous Programming

Due to the constant increase in the diversity of accelerators, development tools must, to one degree or another, ensure the portability of applications between devices. This can manifest itself in two forms: either an already compiled program is launched on different platforms without recompiling, or the program is compiled from one source code for different platforms. The second method also includes compiling platform-dependent fragments into dynamic libraries for different platforms with the subsequent selection of the appropriate fragment for a particular platform.

Porting without recompilation requires a unified intermediate language, and porting with recompilation requires a unified high-level API.

2.1 Portable Intermediate Representation

By the end of the 2000s, the LLVM ecosystem reached a stable state [1]. LLVM is based on the idea that some program translation stages are independent of a programming language or platform, which means that they can be processed by one tool. LLVM provides the middle layers of a complete compiler system, taking intermediate representation (IR) code from a compiler and emitting an optimized IR. This new IR can then be converted and linked into machine-dependent assembly language code for a target platform. It also provides various tools to facilitate the compiler development. The development of this ecosystem has led to the creation of a clang compiler for C and C++ languages, virtual machines for various executable languages, and the emergence of many other tools.

Heterogeneous computing development tools developers could not ignore this approach, and in 2011 the Standard Portable Intermediate Representation (SPIR), an intermediate representation language based on LLVM IR, was released as an optional part of OpenCL 1.2 [2]. It was assumed that the development tools would generate a SPIR code independent of the platform, and the graphic cards drivers and other devices would consume it, transforming it into their own representation. This would make it relatively easy to create compilers in SPIR based on existing implementations in the LLVM ecosystem, since the differences between SPIR and LLVM IR are not too big. In particular, it would be possible to compile OpenCL in SPIR, which would remove OpenCL support from drivers, bringing it to the level of regular libraries.

However, in practice, NVIDIA developers did not provide SPIR support at all, and AMD developers removed SPIR support from drivers for most graphic cards in 2018. As the result, only Intel provides good support for SPIR. Most likely, the refusal to support SPIR was due to the fact that it does not always allow to take into account the features of specific architectures of video cards due to the LLVM IR heritage.

In 2015, the standard SPIR-V was released as a part of the Vulkan ecosystem [2]. It was originally developed taking into account the features of the architecture of high-parallel accelerators, and, perhaps, the developers of such accelerators in the future will provide its full support. However, at the moment they provide only a limited subset of SPIR-V for graphic shaders.

Despite the problems, an ecosystem is still forming around SPIR. The Portable Computing Language (POCL) project provides open OpenCL drivers with SPIR support for CPUs and NVIDIA (experimentally), and there are projects that generate SPIR.

It is also worth noting that NVIDIA has its own open standard for intermediate representation - PTX. It can be used only for NVIDIA GPUs, but serves the same purpose - it allows to transfer applications between different accelerators from NVIDIA.

Thus, the idea of an intermediate representation of programs for accelerators is being developed, but in practice, the ability to compile the same program for different platforms is more important than portability of an already compiled program. From this point of view, standards providing a high-level APIare more interesting.

2.2 High Level API

The most portable API is still OpenCL. Although the degree of support varies for different platforms and many platforms still do not support OpenCL 2.0, it is difficult to find a platform oriented at accelerating parallel data processing but not supporting OpenCL at all. In the case of mobile platforms (especially on Android), there are essentially no portable OpenCL alternatives.

Despite the good portability, OpenCL has many disadvantages. For example, because of the compilation at runtime, the device driver must contain an almost complete C compiler, and the source code of the kernels is open. The kernel works in its own context, even if it runs on a host, which requires unnecessary copying of data from the host into this context. Computing kernels have to be written in a special language that is a subset of C. It creates potential vulnerabilities in data transfer between the host and the device that are not detected by static analyzers. It also means that compiler cannot perform context-sensitive optimization. Apple even declaredOpenCL deprecated in IoS 12 [3].

Many developers prefer to use NVIDIA CUDA. CUDA is free of many of the drawbacks of OpenCL, in particular, it supports the use of C++ (including C++ 17) in the kernels and does not separate the kernel code from the host code, which allows data transfer to be typesafe. The question of comparing the performance of CUDA and OpenCLkernels remains open in general, but cuBLAS (linear algebra library as part of CUDA) outperforms both user made implementations on OpenCL and CUDA, which is also an argument in favor of CUDA [4].

But the choice of CUDA over time can lead to complications: the computing accelerators ecosystem is evolving, new hardware solutions are emerging, and switching to other platforms may become a necessity. For large projects, this can be a really serious problem.

An interesting solution is offered by the ROCm project: it contains a HIP – tool that allows you to semi-automatically convert CUDA code into portable C++ code and compile this portable C++ for AMD and NVIDIA, or you can write initially portable applications. At the moment this tool is only applicable for NVIDIA and AMD, although the idea can be implemented for other platforms [5]. For projects designed for long-term support, standardized APIs originally designed as portable are more interesting.

Such API can be provided by the Parallelism project in the C++ standardization framework. Its idea is that the interfaces provided by the standard C++ library for algorithms are already well suited for parallel algorithms, an example of which is Boost. Compute and other similar libraries. So this interface can be extended for automatic parallelization [6]. The CPU-oriented part of this project is already included in the C++ 17 standard, and there is a potential for expanding the approach to the GPUs and other devices [7]. The main problem here is that the level of abstraction is too high to allow you to manually optimize the code for specific devices. Even the choice of the device is actually inaccessible to the user, the compiler chooses it. But thanks to this project, the C++ memory model is now expanding towards greater support for parallel algorithms, including for accelerators.

A beter compromise option is the SYCL standard - a heterogeneous-oriented API for C++, released in 2014.

2.3 SYCL

SYCL is a high-level heterogeneous programming standard that provides OpenCL abstractions for C++ [6, 8]. One of the main goals of SYCL is to increase the reliability of applications and ensure the portability of applications between different devices. Unlike OpenCL, the code for the host and the accelerator is not separated in the SYCL, which allows you to fully use the C++ type system to ensure secure data transfer between devices, as well as to minimize the overhead in the case of running the computing kernel on the host. In addition, SYCL provides interfaces based on the ideas of modern C++ and the capabilities of C++ 11, which also improves the reliability of programs. In most cases, the resource leak problem no longer arises or can be resolved relatively easily.

The lack of separation of host and accelerator code has other consequences. First of all, the possibilities for optimization are increase: the SYCL compiler has access to the full context of the program, which allows it to perform high-level optimizations, for example, to delete unused code. This approach also makes it much easier to develop tools for SYCL. In a simple case, the program on the SYCL can be compiled by a conventional C++ compiler for the CPU, which allows you to debug it using conventional means.

To run on the accelerator, the program must be compiled by two compilers: the first one finds and compiles the compute kernels, and the second one compiles the host code

and creates an intermediate layer that selects the desired kernel implementation for a particular device. Ideally, one compiled program can contain compiled kernels for CPU, NVIDIA, AMD, and other devices. Or, it can contain a single intermediate view code, such as SPIR.

Compiling kernels for accelerators still needs a separate compiler, but it is relatively easy to implement based on existing tools. Clang is already able to compile C++ into an abstract syntax tree, and the interpretation of this tree is quite a doable task (in comparison opposed to developing a full-featured C++ compiler).

Since this standard, at first glance, has advantages over CUDA and OpenCL, it was chosen for more detailed analysis.

2.4 SYCL Implementations

There are already several implementations of SYCL: ComputeCpp, triSYCL, hipSYCL, sycl-gtx (non-standard macros required), clang (effort announced by Intel on 2019/01/11) [9]. Most of them are based on the LLVM ecosystem and the Clang compiler, but the implementation details and readiness of the implementations are various.

The most "advanced" can be called trySYCL – in fact it is a platform for experiments with the specification of SYCL [10]. This is a fully open source project that can be used to compile the SYCL program for the CPU and, theoretically, for SPIR – compatible accelerators. There are also relatively successful attempts to use it for FPGA [11]. In practice, the deployment of all the necessary tools is a non-trivial task, and in general, the project is more interesting for those who want to participate in the development of the SYCL ecosystem.

The most stable project is ComputeCpp [12]. In the fall of 2018, its developers were the first to declare full support for the SYCL 1.2 specification, providing tools for compiling the SYCL in SPIR and experimental support for SPIR-V. In addition, ComputeCpp allows you to translate SYCL to PTX, which allows you to use it for NVIDIA graphics cards. Support for PTXis not yet complete, but most of the functions are already working. When using old drivers, you can also use AMD graphics cards, but new drivers do not support SPIR for most platforms, and older drivers require older Linux kernels.

The prospects for using ComputeCpp for mobile platforms are also interesting. ComputeCpp already provides development tools for ARM, but so far only for Linux. Perhaps in the future there will be support for Android. ComputeCpp also provides support for R-Car H3 (computer systems for cars).

The main disadvantage of ComputeCpp is closed source code, but it does not impose restrictions on commercial use.

Another promising implementation is HipSycl [9]. This implementation is based on the already mentioned HIP tool, which allows you to compile code for both NVIDIA and AMD. In essence, this is just a library that implements SYCL over an existing API. This is an open source project and is available for commercial use, but at the moment there is no compliance with the SYCL specification, in particular, there is no support for profiling. But in general, it is already operational.

2.5 Other Approaches

All the previously mentioned solutions are focused on C and C++. For the most part, this is due to historical reasons, as well as the desire to optimize programs for accelerators. Nevertheless, development tools that are entirely focused on other languages are being developed.

There are many solutions for .NET, most of which are based on translating C# code into CUDA code on the fly. For example, Aleagpu [13] and Hybridizer [14]. There are similar solutions for Java and other languages. There are experimental projects for translating various languages to SPIR-V [15], for example, SpirvNet for .NET. This can lead to a fragmentation of the ecosystem of computing on accelerators, as has happened with other areas.

Evaluating the effectiveness of using different languages for computing on GPU is beyond the scope of this article, but Rust language is worth mentioning separately [16]. For long-term calculations, the possibility of errors in the program becomes a big problem. In the case of accelerators, this is complicated by inability to handle errors directly in the kernel code, which leads to asynchronous errors on the host side. Such errors are difficult to handle, and often they cannot be fixed without emergency termination of the program. Thus, the issue of finding errors during compilation becomes particularly important.

The C++ type system often allows you to find errors at compile time, which distinguishes it from other languages. But the C++ type system is too complex and contains many problems caused by backward compatibility. Rust uses similar approach, but it was originally designed to find the maximum number of errors during compilation, and it lacks many of the problems inherent in C++. At the same time, it allows to implement zero-cost abstractions as well as C++. All this makes Rust a good candidate for computing in general and for heterogeneous programming in particular.

The advantages of the Rust type system can be seen now, but the GPU computing ecosystem for Rust is in a very early stage of development.

3 SYCL Health and Performance Testing

When it comes to acceleration due to data parallelization, the first thing that comes to mind is the matrix-matrix multiplication and the matrix-vector multiplication operations. These operations were implemented for testing. These operations are convenient for testing, since the results obtained can be compared with the results of CUDA testing in [17].

Since on small data volumes the cost of data transmission to the accelerator exceeds the benefit from acceleration, a set of 64 pairs of matrices of the same size was used for testing. Such a set of 2048×2048 matrices of double precision numbers takes up 4 GB of memory, and another 2 GB is needed to store the result.

To test performance, several classic approaches to accelerating matrix multiplication on graphics cards were implemented. These approaches can be divided into two groups: management of the computation process by the host (optimization of data

transfer between the host and the device) and calculations (optimization of data transmission within the device).

Also for comparison, Eigen is one of the best libraries of linear algebra for the central processor, and similar tests for CUDA are used.

3.1 Ways to Manage the Process of Computing

The following approaches to managing the calculation process have been implemented:

Simple Loop. A simple loop on a set of matrices, at each iteration, a pair of source matrices is transmitted to the device, processed on it, and the result is transmitted to the host. All tasks are performed sequentially. Obviously, for small matrices overhead must be greater than for large ones.

One-In One-Out. All source matrices are written to one large buffer, the result is also written to one buffer, the computational kernels are started asynchronously and can be executed in parallel. Overhead costs should be less dependent on the size of the matrices, but in practice the use of this approach is rarely possible.

One-In Many-Out. The original matrices are also recorded in one buffer, which is transferred to the accelerator once, but the computational kernel is started and the data is transferred to the host sequentially for each pair of initial matrices (the kernels are executed sequentially). This allows the program to perform calculations and transmit to the host the results of previous calculations in parallel.

Auto Transfer Simple Loop. Similar to Simple Loop, but instead of explicit data transfer, SYCL data automatic control was used. In this case, only the total operation time is measured, since SYCL does not measure data transfer times separately with automatic control.

3.2 Ways to Optimize Computational Kernels

Computational kernels are implemented for the operation of matrix multiplication and matrix multiplication by a vector.

Simple Kernel. The classic loop used to multiply matrices by the CPU. The only optimization is to change the traversal order so as to streamline the memory access.

Local Memory Kernel. Classic block algorithm taken from CUDA documentation [18]. At each step, two fragments of the original matrices are placed in the local memory of the working group (shared memory in terms of CUDA), the intermediate result is calculated and also placed in the local memory. In the case of multiplication by a vector, the local memory is not needed for the left matrix. The block size was calculated automatically based on accelerator characteristics.

Private Memory Kernel. It is similar to Local Memory Kernel, but the intermediate result is placed directly into private (register) memory.

3.3 Platform for Testing

The following hardware configuration was used:
CPU: Intel (R) Xeon (R) E5440 @ 2.83 GHz, 64bit, 8 cores
GPU: GeForce GTX 1060 6 GB
max compute units: 10
max work group size: 1024
local memory size: 48 K
CUDA 9.2.106

3.4 Test Results

It was possible to use only one implementation - ComputeCPP. HipSycl failed to install on target machine. Despite the presence of the Docker image, the distributed HipSycl components turned out to be sensitive to the version of the operating system. Theoretically, HipSycl can be compiled from source files, but it has many dependencies, and their resolution was impossible in our conditions.

Test results are visualized in the form of graphs of the dependence of the execution time on the matrix size (N), and the execution time is divided by the number of elements in the left matrix (N ^ 2). The time scale is logarithmic, the lower the better.

The size of the left matrix varied from 1 × 1 to 2048 × 2048.

To begin with, let's compare different implementations of computing kernels with a simple data transfer strategy on device for matrix and matrix multiplication(see Fig. 1) and matrix and vector multiplication (see Fig. 2).

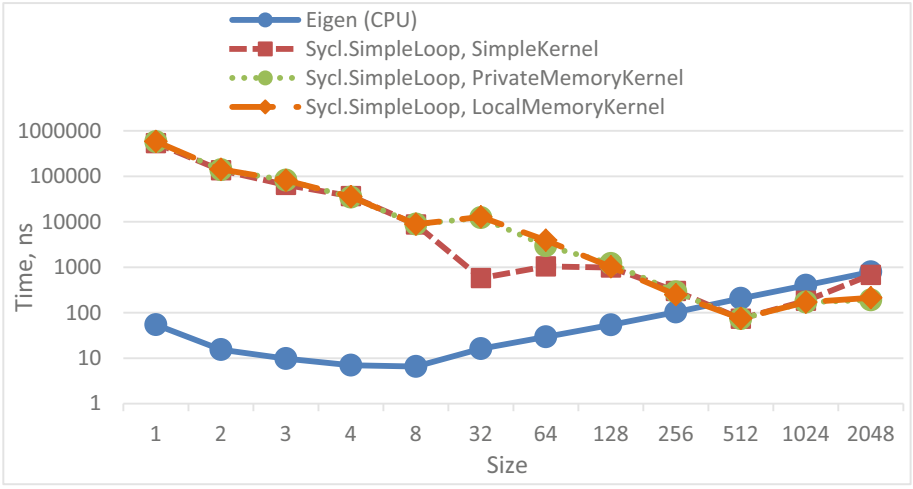

Fig. 1. Test results of a simple data transfer strategy for matrix-matrix multiplication

Surprisingly, none of the optimization yielded a significant performance increase compared with the implementation on the CPU, and the naive implementation on the

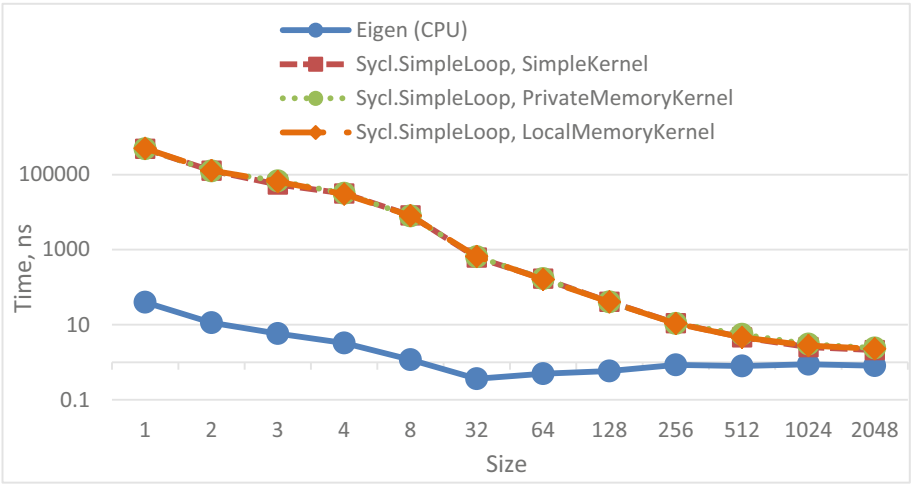

Fig. 2. Test results of a simple data transfer strategy for matrix-vector multiplication

SYCL turned out to be noticeably faster for 32 × 32 and 64 × 64 matrices, which coincides with the restriction on the size of the working group. The optimizer was good enough to compete with typical manual optimizations, but did not provide good overall performance.

What is even more surprising, it was not possible to obtain a significant superiority over the implementation on the CPU, even for large matrices. If for small matrices this result is understandable, then for large ones it raises questions.

Since there are no significant differences in the performance of computational kernels, we tried to compare different ways of managing computations using a simple kernel for matrix-matrix multiplication (see Fig. 3) and matrix-vector multiplication (see Fig. 4).

The first conclusion that can be made that when using automatic data transfer, time increases by an order of magnitude. This is an obvious implementation problem, since the rules for applying automatic data transfer are sufficiently transparent so that it does not differ from the manual one.

In general, the applied optimizations gave a positive result, but the implementation of Eigen for the CPU is still ahead of them all. One-In One-Out optimization surpasses the rest for small matrices, but for large ones One-In Many-Out or even Simple Loop is more efficient.

Since the Eigen implementation is still ahead of all optimizations for SYCL, it makes sense to compare the implementations for SYCL and CUDA for the matrix-vector multiplication (see Fig. 5).

The implementation for CUDA also did not receive a significant advantage over the implementation of Eigen for the CPU, which can be explained by the active use of SIMD - instructions in the implementation of Eigen and a relatively weak graphics card, but compared to SYCL this implementation is 1–3 orders of magnitude faster.

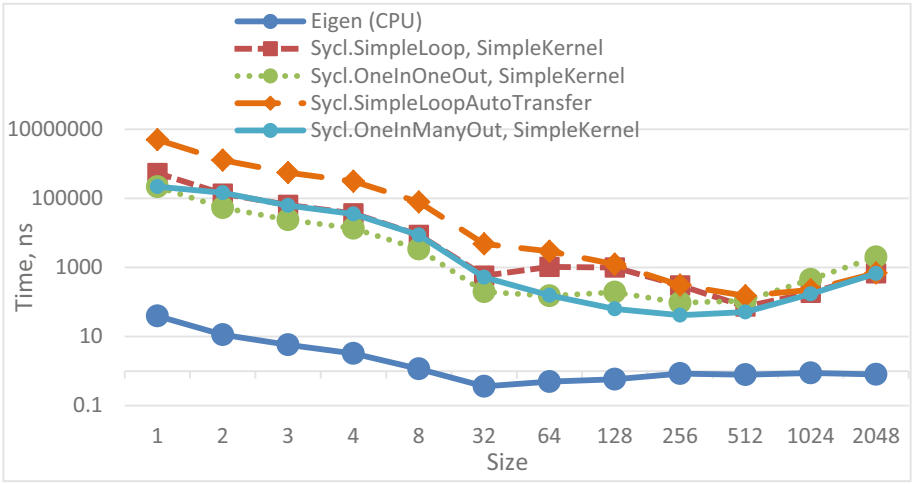

Fig. 3. Test results with different data transfer strategies for matrix-matrix multiplication

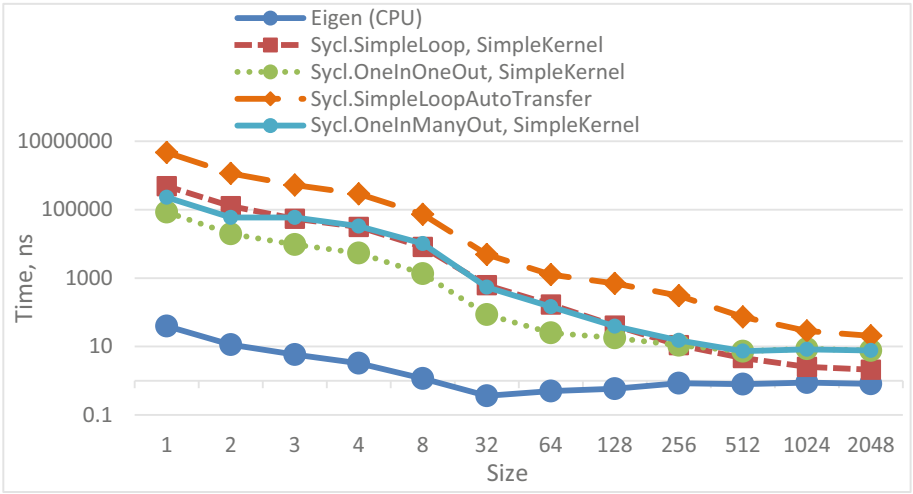

Fig. 4. Test results with different data transfer strategies for matrix-vector multiplication

To localize the problem, an independent comparison was made between the calculation time(see Fig. 6) and the data transfer time(see Fig. 7).

The data transfer time for SYCL is higher by 3–4 orders of magnitude, and the kernel execution time is only one order of magnitude. The low performance of the kernel is quite understandable, since the developers have added support for PTX only recently. But it is difficult to explain such high costs of data transfer. The implementation should simply call the functions supplied by the driver, and there should be no additional costs.

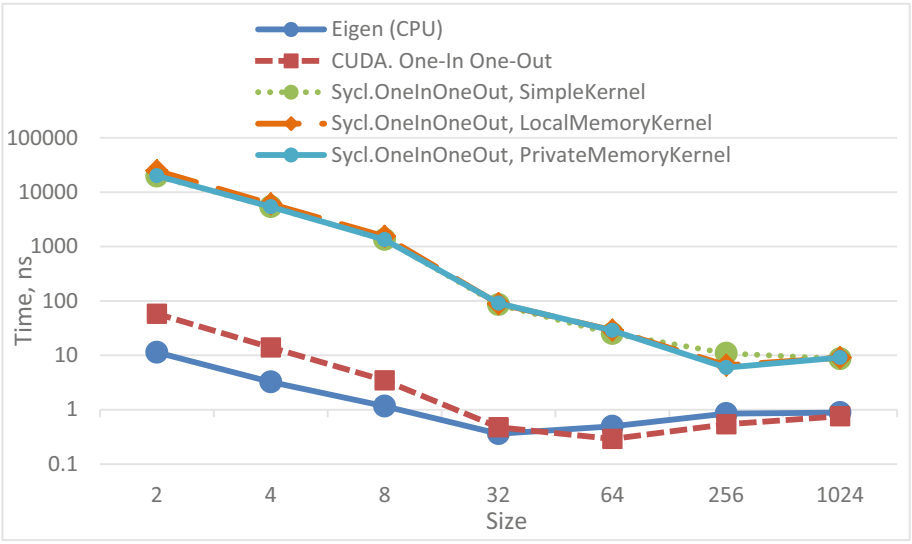

Fig. 5. Results of comparing SYCL with CUDAfor matrix-vector multiplication

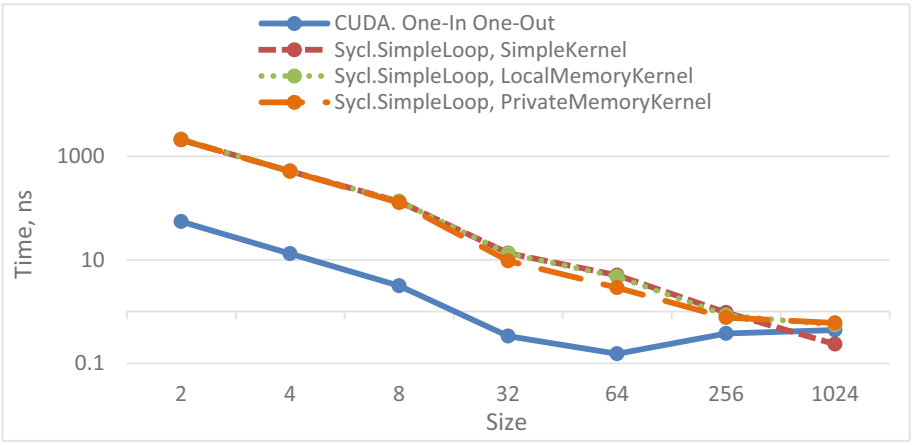

Fig. 6. Calculation time

3.5 Recommendations for Choosing Development Tools for Parallel Computing Accelerators

As shown by the test results, the ComputeCPP implementation does not yet provide sufficient performance for SYCL, at least on NVIDIA graphics cards. The HipSYCL implementation has not been tested, but in any case it does not fully comply with the specification.

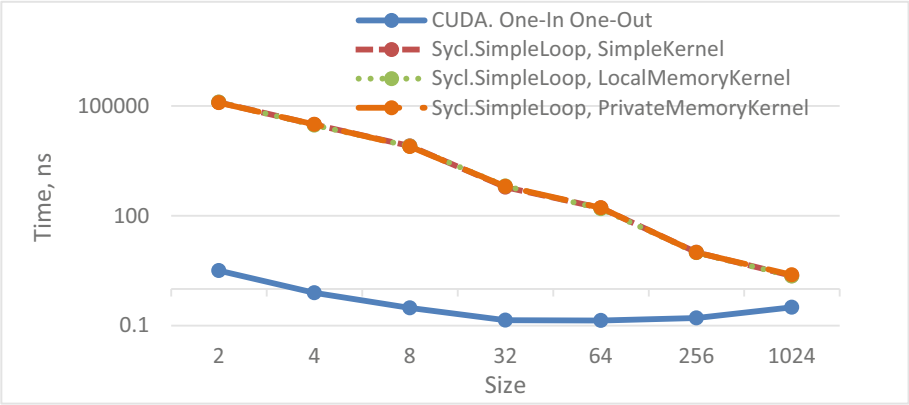

Fig. 7. Transfer Time

Thus, at the moment, the main development tools for heterogeneous computing are still CUDA and OpenCL. However, the development of SYCL is fast, and for long-term projects it makes sense to consider the possibility of switching to SYCL, especially since all these tools are based on similar abstractions.

In some cases, tools for languages other than C and C++ may also be useful. But since they usually translate the code into CUDA or OpenCL, they do not provide significant advantages, apart from the ability to preserve the homogeneity of the development ecosystem.

In addition, one should not discount calculations on the central processor - in many cases, proper use of SIMD instructions (or libraries using them) allows to achieve good performance, which is especially noticeable against the background of reducing data transfer costs.

4 Conclusion

In this paper, we considered promising development tools for portable programs for parallel computing accelerators, in particular, the new standard for heterogeneous programming SYCL and its main implementations.

The ComputeCpp implementation for SYCL has been tested. The results show that this implementation works correctly, but still does not provide sufficient performance for practical use, at least for NVIDIA graphics cards. However, given that this implementation is a beta version, and NVIDIA support was added quite recently, there is hope for rapid changes for the better.

The implementation of HipSycl has not been tested due to installation difficulties, but it looks more suitable for practical use at present. Since it does not compile the source code on its own, but simply wraps the CUDA and HCC implementations into a single interface, there is no reason for a serious decrease in performance compared to the original tools. However, this still requires verification.

In general, SYCLstill leaves a positive impression, if one does not take into account the issue of performance. Modern C++ idioms really allowed as to detect many errors during compilation. However, for programmers who are new to C++, this can be a disadvantage, since the causes of compilation errors are not always obvious.

References

1. LLVM. https://llvm.org. Accessed 31 Mar 2019
2. SPIR Overview. https://www.khronos.org/spir. Accessed 31 Mar 2019
3. The end of OpenGL support, plus other updates Apple didn't share at the keynote. https://arstechnica.com/gadgets/2018/06/the-end-of-opengl-support-other-updates-apple-didnt-share-at-the-keynote. Accessed 31 Mar 2019
4. Tutorial: OpenCL SGEMM tuning for Kepler. https://cnugteren.github.io/tutorial/pages/page1.html. Accessed 31 Mar 2019
5. ROCm project page. https://github.com/RadeonOpenCompute/ROCm. Accessed 31 Mar 2019
6. Keryell, R., Reyes, R., Howes, L.: Khronos SYCL for OpenCL. In: Proceedings of the 3rd International Workshop on OpenCL – IWOCL 2015 (2015). https://doi.org/10.1145/2791321.2791345
7. Parallel Algorithms for Better Performance. https://devblogs.microsoft.com/cppblog/using-c17-parallel-algorithms-for-better-performance. Accessed 31 Mar 2019
8. SYCL. https://www.khronos.org/sycl. Accessed 31 Mar 2019
9. hipSYCL project page. https://github.com/illuhad/hipSYCL. Accessed 31 Mar 2019
10. triSYCL project page. https://github.com/triSYCL/triSYCL. Accessed 31 Mar 2019
11. Keryell, R., Yu, L.Y.: Early experiments using SYCL single-source modern C++ on Xilinx FPGA. In: Proceedings of the International Workshop on OpenCL – IWOCL 2018 (2018). https://doi.org/10.1145/3204919.3204937
12. Codeplay: ComputeCpp. https://www.codeplay.com/products/computecpp. Accessed 31 Mar 2019
13. Alea GPU. http://www.aleagpu.com/release/3_0_3/doc/introduction.html. Accessed 31 Mar 2019
14. Hybridizer. http://www.altimesh.com. Accessed 31 Mar 2019
15. SpirNet project page. https://github.com/Philip-Trettner/SpirvNet. Accessed 31 Mar 2019
16. Rust (programming language). https://en.wikipedia.org/wiki/Rust_(programming_language). Accessed 31 Mar 2019
17. Storublevtcev, N., Korkhov, V., Beloshapko, A., Bogdanov, A.: Application Porting Optimization on Heterogeneous Systems. In: Gervasi, O., et al. (eds.) ICCSA 2018. LNCS, vol. 10963, pp. 25–40. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95171-3_3
18. CUDA C Best Practices Guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#shared-memory. Accessed 31 Mar 2019