The Journal of Supercomputing
https://doi.org/10.1007/s11227-024-05907-2

Assessing opportunities of SYCL for biological sequence alignment on GPU-based systems

Manuel Costanzo[1] · Enzo Rucci[1] · Carlos García-Sanchez[2] · Marcelo Naiouf[1] · Manuel Prieto-Matías[2]

Accepted: 5 January 2024

Abstract
Bioinformatics and computational biology are two fields that have been exploiting GPUs for more than two decades, with being CUDA the most used programming language for them. However, as CUDA is an NVIDIA proprietary language, it implies a strong portability restriction to a wide range of heterogeneous architectures, like AMD or Intel GPUs. To face this issue, the Khronos group has recently proposed the SYCL standard, which is an open, royalty-free, cross-platform abstraction layer that enables the programming of a heterogeneous system to be written using standard, single-source C++ code. Over the past few years, several implementations of this SYCL standard have emerged, being oneAPI the one from Intel. This paper presents the migration process of the *SW#* suite, a biological sequence alignment tool developed in CUDA, to SYCL using Intel's oneAPI ecosystem. The experimental results show that *SW#* was completely migrated with a small programmer intervention in terms of hand-coding. In addition, it was possible to port the migrated code between different architectures (considering multiple vendor GPUs and also CPUs), with no noticeable performance degradation on five different NVIDIA GPUs. Moreover, performance remained stable when switching to another SYCL implementation. As a consequence, SYCL and its implementations can offer attractive opportunities for the bioinformatics community, especially considering the vast existence of CUDA-based legacy codes.

Keywords SYCL · OneAPI · GPU · CUDA · SYCLomatic · Bioinformatics · DNA · Protein · Sequence alignment

1 Introduction

Hardware specialization has consolidated as an effective way to continue scaling performance and efficiency after Moore's law ended. Compared to CPUs, hardware accelerators can offer orders of magnitude improvements in performance/cost and

Extended author information available on the last page of the article

Published online: 19 February 2024

performance/W [1]. That is the main reason why the programmers typically rely on a variety of hardware, such as GPUs (Graphics Processing Units), FPGAs (Field-programmable Gate Array), and other kinds of accelerators, (e.g., TPUs), depending on the target application. Unfortunately, each kind of hardware requires different development methodologies and programming environments, which implies the usage of different models, programming languages, and/or libraries. Thus, the benefits of hardware specialization come at the expense of increasing the programming costs and complexity and complicating future code maintenance and extension.

In this context, GPUs are present in the vast majority of high performance computing (HPC) systems and CUDA is the most used programming language for them [2]. Bioinformatics and computational biology are two fields that have been exploiting GPUs for more than two decades [3]. Many GPU implementations can be found in sequence alignment [4], molecular docking [5], molecular dynamics [6], and prediction and searching of molecular structures [7], among other application areas. However, as CUDA is an NVIDIA proprietary language, it implies a strong portability restriction to a wide range of heterogeneous architectures. To take a case in point, CUDA codes cannot run on AMD or Intel GPUs.

In the last decades, academia and companies have been working on developing a unified language to program heterogeneous hardware, capable of improving productivity and portability. Open Computing Language (OpenCL) [8] is a standard maintained by the Khronos group, which has facilitated the development of parallel computing programs for execution on CPUs, GPUs, and other accelerators. Even though OpenCL is a mature programming model, an OpenCL program is much more verbose than a CUDA program and its development tends to be tedious and error-prone [9]. That is why the Khronos group has recently proposed the SYCL standard,[1] which is an open, royalty-free, cross-platform abstraction layer that enables the programming of a heterogeneous system to be written using standard, single-source C++ code. Moreover, SYCL sits as a higher level of abstraction, offering backend implementations that map to contemporary accelerator languages, like CUDA, OpenCL, and HIP.

Currently, several implementations follow the SYCL standard and Intel's oneAPI is one of them. The core of oneAPI programming ecosystem is a simplified language for expressing parallelism on heterogeneous platforms, named Data Parallel C++ (DPC++), which can be summarized as C++ with SYCL. In addition, oneAPI also comprises a runtime, a set of domain-focused libraries, and supporting tools [10].

Due to the vast existence of CUDA-based legacy codes, oneAPI includes a compatibility tool (`dpct` renamed as `SYCLomatic`) that facilitates the migration to the SYCL-based DPC++ programming language. In this paper, we present our experiences porting a biological software tool to DPC++ using `SYCLomatic`. In particular, we have selected *SW#* [11]: a CUDA-based, memory-efficient implementation for biological sequence alignment, which can be used either as a stand-alone application or a library. This paper is an extended and thoroughly revised version of [12]. The work has been extended by providing:

[1] https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf.

- The complete migration of SW# to SYCL (not just the package for protein database search). This code represents a SYCL-compliant, DPC++-based version of SW# and is now available at a public git repository.[2]
- An analysis of the efficiency of the `SYCLomatic` tool for the CUDA-based SW# migration, including a summary of the porting steps that required manual modifications.
- An analysis of the DPC++ code's portability, considering different target devices and vendors (NVIDIA GPUs; AMD GPUs and CPUs; Intel GPUs and CPUs), and SW# functionalities. Complementing our previous work, we have considered 5 NVIDIA GPU microarchitectures, 3 Intel GPU microarchitectures, 2 AMD GPU microarchitectures, 4 Intel CPU microarchitectures, and 1 AMD CPU microarchitecture. In addition, the analysis includes both DNA and protein sequence alignment, in a wide variety of scenarios (alignment algorithm, sequence size, scoring scheme, among others). Moreover, cross-SYCL-implementation portability is also verified on several GPUs and CPUs.
- A comparison of the performance on the previous hardware architectures for the different biological sequence alignment operations that were considered.

The remaining sections of this article are organized as follows. Section 2 explains the background required to understand the rest of the article and Sect. 3 describes the migration process and the experimental work carried out. Next, Sect. 4 presents the experimental results and discussion. Finally, Sect. 5 concludes the paper.

2 Background

2.1 Biological sequence alignment

A fundamental operation in bioinformatics and computational biology is sequence alignment, whose purpose is to highlight areas of similarity between sequences to identify structural, functional, and evolutionary relationships between them [4].

Sequence alignment can be global, local, or semi-global. Global alignment attempts to align every residue of every sequence and is useful when sequences are very similar to each other. Local alignment is better when the sequences are different but regions of similarity between them are suspected. Finally, semi-global alignment is based on the global alternative, with the difference that it seeks to penalize internal gaps, but not those found at the beginning or end of any of the sequences [13].

Any of these algorithms can be used to compute: (a) pairwise alignments (one-to-one); or (b) database similarity searches (one-to-many). Both cases have been parallelized in the literature. In case (a), a single matrix is calculated and all processing elements (PEs) work collaboratively (*intra-task parallelism*). Due to inherent data dependencies, neighboring PEs communicate to exchange border elements. In case (b), while intra-task scheme can be used, a better approach consists in

[2] https://github.com/ManuelCostanzo/swsharp_sycl.

simultaneously calculating multiple matrices without communication between the PEs (*inter-task parallelism*) [4].

2.1.1 Needleman–Wunsch algorithm (NW)

In 1970, Saul Needleman and Christian Wunsch proposed a method for aligning protein sequences [14]. It is a typical example of dynamic programming which guarantees that the optimal global alignment is obtained, regardless of the length of the sequences, and presents quadratic time and space complexities.

2.1.2 Smith–Waterman algorithm (SW)

In 1981, Smith and Waterman [15] proposed an algorithm to obtain the optimal local alignment between two biological sequences. SW maintains the same programming model and complexity as NW. Furthermore, it has been used as the basis for many subsequent algorithms and is often employed as a benchmark when comparing different alignment techniques [16]. Unlike global alignments, local alignments consider the similarity between small regions of the two sequences, which usually makes more biological sense [17].

2.1.3 Semi-global algorithm (HW)

A semi-global alignment does not penalize gaps at the beginning or end in a global alignment, so the resulting alignment tends to overlap one end of one sequence with one end of the other sequence [18].

2.1.4 Overlap algorithm (OV)

An overlap of two sequences is an alignment in which the initial and final gaps are ignored. It is considered a variant of the semi-global alignment because the two sequences are aligned globally but without taking into account the end gaps at both ends [19].

2.2 SW# suite

SW# is a software released in 2013 for biological sequence alignment. It can compute pairwise alignments as well as database similarity searches, for both protein and DNA sequences [20]. This software allows configuring the algorithm to be used for different alignments (SW, NW, HW, OV) as well as open/extension penalties, and also the substitution matrix (BLOSUM45, BLOSUM50, BLOSUM62, among others, for proteins; and match/mismatch values for DNA). As it combines CPU and GPU computation, it allows configuring the number of CPU threads and GPU devices to be used.

SW# applies specific CPU and GPU optimizations, which significantly reduce the execution time. On the CPU side, SW# uses the OPAL.[3] library that allows optimizing the search for sequence similarities using the Smith–Waterman algorithm, through the use of multithreading and SIMD instructions. On the GPU side, SW# follows both inter-task and intra-task parallelism approaches (depending on the sequence length) [11]

In particular, SW# has developed its efficient version of SW algorithm. This algorithm can be divided into two phases: resolution and reconstruction. In the resolution phase, the maximum score is calculated, while in the reconstruction phase, the optimal alignment path is obtained. For this second stage, SW# uses space-efficient methods [21].

2.3 Hardware accelerators

Hardware acceleration aims to increase the performance and the energy efficiency of applications by combining the flexibility of general-purpose processors, such as CPUs, with the potential of specific hardware, called hardware accelerators. The most used accelerators in HPC are GPUs. Although they were initially designed to speed up graphic rendering, GPUs have been used in general scientific contexts due to the massive incorporation of computing units. Historically, NVIDIA and AMD have been the manufacturers, while Intel recently joined as a competitor.

2.3.1 CUDA

In 2006, NVIDIA introduced CUDA (Compute Unified Device Architecture), a new architecture containing hundreds of processing cores or CUDA cores. CUDA is an extension of C/C++ and provides an abstraction of the GPU, acting as a bridge between the CPU and GPU [22]. However, CUDA is a proprietary language that only runs on NVIDIA GPUs, which limits code portability.

2.3.2 SYCL and its implementations

SYCL is a cross-platform programming model based on C++ language for heterogeneous computing, announced in 2014. It is a cross-platform abstraction layer that builds on the underlying concepts, efficiency, and portability inspired by OpenCL,[4] which allows the same C++ code to be used on heterogeneous processors.

Nowadays, multiple SYCL implementations are available: Codeplay's ComputeCpp [23] (now part of oneAPI[5]), oneAPI by Intel [24], triSYCL [25] led by Xilinx, and AdaptiveCpp [26] (previously denoted as hipSYCL/OpenSYCL [27]) led by Heidelberg University. In particular, Intel oneAPI can be considered the most mature developer suite. It is an ecosystem that provides a wide variety of

[3] https://github.com/Martinsos/opal.

[4] https://www.khronos.org/opencl/.

[5] https://codeplay.com/portal/news/2023/07/07/the-future-of-computecpp.

development tools across different devices, such as CPUs, GPUs, and FPGAs. OneAPI provides two different programming levels: on the one hand, it supports direct programming through Data Parallel C++ (DPC++), an open, cross-platform programming language that offers productivity and performance in parallel programming. DPC++ is a fork of the Clang C++ and incorporates SYCL for heterogeneous programming while containing language-specific extensions. On the other hand, it supports API-based programming, by invoking optimized libraries (such as oneMKL, oneDAL, oneVPL, etc.). Within its variety of programming utilities, oneAPI offers `SYCLomatic`, a tool to convert code written in CUDA to SYCL.[6]

AdaptiveCpp [26] is a platform that facilitates C++-based heterogeneous programming for CPUs and GPUs. It integrates SYCL parallelism, enabling the offloading of C++ algorithms to a wide range of CPU and GPU vendors (such as Intel, NVIDIA, and AMD). AdaptiveCpp applications can dynamically adapt to diverse hardware. In particular, a single binary can target various hardware or even concurrent hardware from different vendors. This is enabled by a new feature of AdaptiveCpp (denoted as *generic single-pass* [28]) that increases the portability and productivity by hiding the dependency on the target hardware. Specifically, AdaptiveCpp employs a generic, single-source, single compiler pass flow (SSCP), compiling kernels into a generic LLVM IR representation. At runtime, this representation is transformed into backend-specific formats like PTX or SPIR-V as required. This approach involves a single compiler invocation, parsing the code once, regardless of the number of devices or backends used. Even so, AdaptiveCpp allows the developer to indicate the specific toolchain/backend compilation flow (if preferred).

3 Materials and methods

In this section, we describe the migration process to reach a SYCL-compliant, DPC++-based version of SW#. Next, we detail the experimental work carried out to analyze the SYCL code's portability and performance.

3.1 Migration process

Generally, `SYCLomatic` is not capable of generating a final code ready to be compiled and executed. It is necessary to perform some hand-tuned modifications to the migrated code, taking advantage of the warnings and recommendations provided by the tool.[7] These warnings vary between aspects of the device to be taken into account (e.g., not to exceed the device's maximum number of threads), modifications to improve performance or even incompatible code fragments. Fortunately,

[6] SYCLomatic: A New CUDA*-to-SYCL* Code Migration Tool: https://www.intel.com/content/www/us/en/developer/articles/technical/syclomatic-new-cuda-to-sycl-code-migration-tool.html.

[7] Diagnostics Reference of Intel® DPC++ Compatibility Tool available at: https://software.intel.com/content/www/us/en/develop/documentation/intel-dpcpp-compatibility-tool-user-guide/top/diagnostics-reference.html.

SYCLomatic reports warnings through an error code along with a description of the issue, within the source code.

The migration process can be divided into 5 stages: (1) running the SYCLomatic tool to generate the first version of the code, (2) modifying the migrated code based on SYCLomatic warnings to obtain the first executable version, (3) fixing runtime errors to obtain the first functional version, (4) verifying the correctness of the results, and (5) optimizing the resulting code, if necessary.

3.1.1 Compilation errors and warnings

After obtaining the first migrated version, the following warnings were reported by SYCLomatic:

- DPCT1003 - Migrated API does not return error code. (*, 0) is inserted. You may need to rewrite this code: this is a very common warning in SYCLomatic and occurs when using CUDA-specific functions, such as error codes.
- DPCT1005 - The SYCL device version is different from CUDA Compute Compatibility. You may need to rewrite this code: this is because the original code is querying for CUDA-specific features, which would not make sense on another device.
- DPCT1049 - The workgroup size passed to the SYCL kernel may exceed the limit. To get the device limit, query info::device::max_work_group_size. Adjust the workgroup size if needed: because the migrated code may run on several devices, SYCLomatic warns not to exceed the maximum capabilities of the devices (e.g., do not exceed the maximum number of threads).
- DPCT1065 - Consider replacing sycl::nd_item::barrier() with sycl::nd_item::barrier (sycl::access::fence_space::local_space) for better performance if there is no access to global memory: SYCLomatic recommends to add a parameter when synchronizing threads as long as global memory is not used.
- DPCT1084 - The function call has multiple migration results in different template instantiations that could not be unified. You may need to adjust the code.: occurs when generic functions are used, which although DPC++ supports it, for the moment SYCLomatic is not able to migrate it.
- DPCT1059 - SYCL only supports 4-channel image format. Adjust the code.: in CUDA it is possible to create texture memories from 1 to 4 channels in SYCL, only 4-channel texture memories (called images in DPC++) can be created and this is alerted by SYCLomatic.

Figure 1 summarizes the warnings generated by SYCLomatic grouped into four areas: error handling (DPCT1003), not supported features (DPCT1005, DPCT1084, and DPCT1059), recommendations (DPCT1049), and optimizations (DPCT1065).

The vast majority (67.1%) is caused due to differences between CUDA and SYCL when handling possible runtime errors.

3.1.2 Code modifications

The following code modifications have been applied to solve the alerts generated by `SYCLomatic`:

- `DPCT1005`: This condition has been removed because there is no equivalent in SYCL.
- `DPCT1049`: the workgroup sizes have been adjusted to the maximum supported by the device.
- `DPCT1065`: the recommendation was followed.
- `DPCT1084`: the use of generic functions has been replaced by conditional sentences that execute the corresponding function.
- `DPCT1059`: the conflicting structures were adapted to four channels.

3.1.3 Runtime errors

At this point, it was possible to compile and execute the migrated code, but the following runtime error was obtained:

$\texttt{Fora1D/2Dimage/imagearray, thewidthmustbeaValue} >= \texttt{1and} <$
$= CL\_\texttt{DEVICE\_IMAGE2D\_MAX\_WIDTH}$

This error appears because the maximum size for image arrays has been exceeded. To solve this issue, the corresponding image array was migrated to the DPC++ unified shared memory (USM)[8]

3.1.4 Code results check

After finishing the migration process, different tests were carried out, both for protein and DNA sequences, using different alignment algorithms and scoring schemes. Finally, it was verified that both CUDA and DPC++ produced the same results.

3.1.5 Code update and tuning

SW# code was designed just for NVIDIA GPUs and is particularly customized for those released in mid-2010. Some configurations are statically indicated in the code, e.g., the block dimensions for kernels. This leads to two limitations when running the migrated code on other devices. First, the code does not take full advantage of current NVIDIA GPUs, which present larger memory capacity and computing power. Second, it prevents execution on devices with different work-group requirements, such as Intel GPUs or CPUs.

[8] https://oneapi-src.github.io/DPCPP_Reference/model/unified-shared-memory.html.

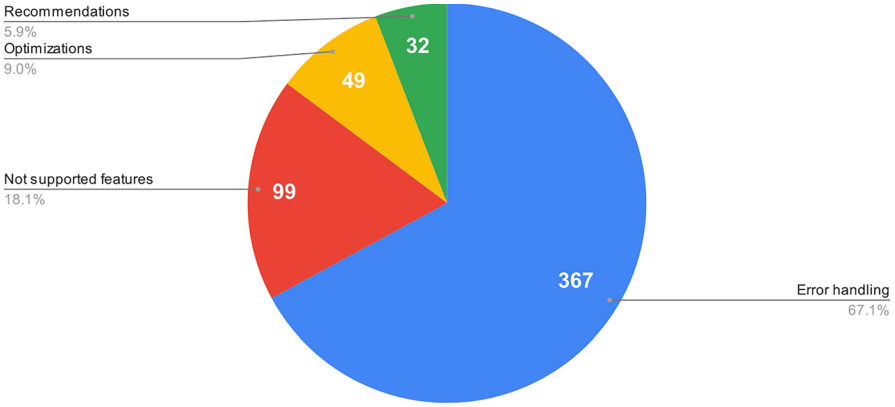

Fig. 1 Distribution of the warnings generated by `SYCLomatic`

To remedy this problem, the static setting of work-group size[9] was replaced by a dynamic configuration that considers the sequence lengths and the maximum allowed value by the corresponding device.[10] In this way, the migrated code support is extended to devices from different architectures.

3.1.6 SYCL standardization (optional)

While the DPC++ language is based on SYCL, it is not fully compliant with the latter. Thus, `SYCLomatic` produces code that depends on the oneAPI ecosystem. For example, in this case, the migrated code declares variables in the constant memory and queries device attributes using DPC++-specific functions. Thus, some manual adjustments must be made to reach a fully compliant SYCL code. On the one hand, the constant memory variables were replaced by kernel arguments, which still reside in constant memory when running on GPUs.[11] On the other hand, DPC++-specific functions were replaced by pure SYCL calls to query the device information. As a result, this final version of the code can be compiled with any of the SYCL-compatible compilers.[12]

[9] A DPC++ work-group is a CUDA block.

[10] It is important to remark that the same enhancement was also applied to the original CUDA code to avoid bias in performance evaluation.

[11] It is important to note that this change implied a significant reduction in the number of lines of code.

[12] Fortunately, several are available from an increasing number of vendors https://www.khronos.org/sycl/.

Table 1 Experimental platforms used in the tests

CPU[1]			GPU[2]			
ID	Processor	RAM (Memory)	ID	Vendor (Type)	Model (Architecture)	GFLOPS Peak (SP)
		16 GB	*GTX 980*	NVIDIA (Discrete)	GTX 980 (Maxwell)	5000
		16 GB	*GTX 1080*	NVIDIA (Discrete)	GTX 1080 (Pascal)	8873
Xeon Gold	Intel Xeon Gold 6138	64 GB	*V100*	NVIDIA (Discrete)	V100 (Volta)	14130
		64 GB	*RTX 3090*	NVIDIA (Discrete)	RTX 3090 (Ampere)	35580
Core-i5	Intel Core i5-7400	8 GB	*RTX 2070*	NVIDIA (Discrete)	RTX 2070 (Turing)	7465
Core-i9	Intel Core i9-9900K	65 GB	*P630*	Intel (Integrated)	UHD Graphics P630 (Gen 9.5)	441.6
	Intel Core i9-13900k	65 GB	*ARC770*	Intel (Discrete)	A770 (Xe HPG)	19660
Xeon E5	Intel Xeon E5-1620	32 GB	*RX6700*	AMD (Discrete)	RX 6700 XT (RDNA2)	13215
Core-i7	Intel Core i7-1165G7	16 GB	*Iris Xe*	Intel (Integrated)	Iris Xe Graphics (Gen 12.1)	1690
Ryzen3	AMD Ryzen 3 5300U	12 GB	*RX Vega 6*	AMD (Integrated)	RX Vega 6 (Vega)	845.6

3.2 Experimental work

All the tests were carried out using the platforms described in Table 1[13]. The oneAPI and CUDA versions are 2023.0.0 and 11.7, respectively, and to run DPC++ codes on NVIDIA GPU, we have built a DPC++ toolchain with support for NVIDIA CUDA, as it is not supported by default on oneAPI.[14] Regarding to AdaptiveCPP, we have used v23.10.0 build from the public repository[15] with clang-v15.0, CUDA v11.7 and ROCm v5.4.3.

For protein alignments, the following databases and configurations were used:

- UniProtKB/Swiss-Prot (Swiss-Prot) database (release 2022_07)[16]: The database contains 204173280 amino acid residues in 565928 sequences with a maximum length of 35213.
- Environmental Non-Redundant (Env. NR) database (release 2021_04)[17]: The database contains 995210546 amino acid residues in 4789355 sequences with a maximum length of 16925.
- The input queries range in length from 144 to 5478, and they were extracted from the Swiss-Prot database (accession numbers: P02232, P05013, P14942, P07327, P01008, P03435, P42357, P21177, Q38941, P27895, P07756, P04775, P19096, P28167, P0C6B8, P20930, P08519, Q7TMA5, P33450, and Q9UKN1).
- The substitution matrix selected is BLOSUM62 and the insertion and gap extension scores were set to 10 and 2, respectively.

For DNA alignments, Table 2 presents the accession numbers and sizes of the sequences used. The score parameters used were +1 for match, −3 for mismatch, −5 for gap open, and −2 for gap extension.

To eliminate the CPU impact on performance, SW# has been configured in GPU-only mode (flag `T=0`). On the other hand, different work-group sizes have been configured to obtain the optimal one. Finally, each test was run twenty times, and performance was calculated as an average to minimize variability.

4 Results and discussion

In this section, we assess the efficiency of the `SYCLomatic` tool for the CUDA-based SW# migration. Next, we analyze the SYCL code's portability and performance, considering different target platforms and vendors (NVIDIA GPUs; AMD

[13] As the original SW# is an *old* CUDA-based software, we tried to include older NVIDIA GPUs (f.e. a Kepler-based one). However, oneAPI only supports NVIDIA GPUs from Maxwell onwards; thus, it was not possible to include them in the performance comparison.

[14] https://intel.github.io/llvm-docs/GetStartedGuide.html.

[15] AdaptiveCpp project: https://github.com/AdaptiveCpp/AdaptiveCpp.

[16] Swiss-Prot: https://www.uniprot.org/downloads.

[17] ENV NR: https://ftp.ncbi.nlm.nih.gov/blast/db/.

GPU; Intel CPUs and GPUs), and SW# functionalities. Last, we discuss the obtained results considering related works.

4.1 SYCLomatic efficiency

In this context, *efficiency* refers to how good is `SYCLomatic` to automatically translate the CUDA code to SYCL. In particular, this issue is evaluated by measuring the SW# source lines of code (SLOC) for CUDA and DPC++ versions[18] (see Table 3). The original SW# version presents 8072 SLOC. After running `SYCLomatic`, we found that 407 CUDA SLOC were not automatically migrated. To reach the first functional DPC++ version, some hand-tuned modifications were required, increasing SLOC to 12175. In summary, `SYCLomatic` succeeded in migrating 95% of the CUDA code, confirming Intel's claims. However, it was necessary to add 1718 SLOC (+21%) to the `SYCLomatic` output to obtain the first executable version. Finally, by removing some `SYCLomatic`-specific code (SYCL standardization), DPC++ SLOC got reduced by approximately 20%.

4.2 Performance and portability results

4.2.1 Performance results

GCUPS (billion cell updates per second) is the performance metric generally used in the SW context [29]. Figure 2 presents the performance of both CUDA and SYCL versions when varying work-group size, using the Swiss-Prot database and the SW algorithm.[19] It is possible to notice that both codes are sensitive to the work-group size; in fact, dynamic configuration obtained the best results for all cases. Moreover, both codes are able to extract more GCUPs when using more powerful GPUs.

Figure 3 complements the previous one by including Env. NR database, whose size is about 7 times bigger than Swiss-Prot. On the one hand, the performance for both versions holds for larger workloads, which turns out to be beneficial for scaling. On the other hand, no code reached the best performance in all cases. The CUDA version showed superiority on the GTX 980 and the GTX 1080 for both databases and also on the V100 and RTX 3090 but only for the Swiss-Prot case. However, it is important to remark that the performance improvement is up to 2% in the best-case scenario. A similar phenomenon occurs on the V100 and RTX 3090 with the Env. NR database, where the SYCL implementation was the fastest one, but just reaching up to 2% higher GCUPS. Last, the performance difference between both codes was smaller than 1% on the RTX 2070. Thus, due to the small performance differences, it can be said that no marked differences can be noted between the two languages for these experiments.

[18] To measure SLOC, the `cloc` tool was used (available at https://github.com/AlDanial/cloc), and blank lines and comments were excluded.

[19] On the GTX 980, it was impossible to compute using block size = 1024 because it exceeds the maximum global memory size of this GPU.

Table 2 DNA sequence information used in the tests

Sequence 1[1]		Sequence 2[2]		
Accession	Size	Accession	Size	Matrix size (cells)
CP000051.1	1 M	AE002160.2	1 M	1 G
BA000035.2	3 M	BX927147.1	3 M	9 G
AE016879.1	5 M	AE017225.1	5 M	25 G
NC_005027.1	7 M	NC_003997.3	5 M	35 G
NC_017186.1	10 M	NC_014318.1	10 M	100 G

The influence of query length can be seen in Fig. 4.[20] First, as expected, a longer query leads to better performance. Second, this chart allows us to further explore what is observed in Fig. 2, showing that although more powerful GPUs have higher performance, a sufficiently large workload is necessary to take advantage of their computing power. For example, the RTX 3090 achieves the best performance but just when the query sequence is longer than 3005 residues.

To avoid the biases of the default configuration, we have considered different alignment algorithms and scoring schemes for the same experiments (see Figs. 5 and 6, respectively). The performance difference for both variants is 2% on average, rising up to 4% in a few cases. Therefore, none of these parameters seems to have an impact on the performance of the migrated code.

Pairwise alignment presents different parallelization challenges to database similarity search. *SW#* employs the inter-task parallelism approach for the former (kernel `swSolveSingle`) and the intra-parallelism scheme for the latter (kernel `swSolveShortGpu`). In consequence, the performance comparison on DNA alignments is presented in Fig. 7. Firstly, contrary to the protein case, longer DNA sequences do not always lead to more GCUPS. This fact can be attributed to the particularities of DNA sequence alignment, such as the degree of similarity between them, as was already observed in [30]. Secondly, the performance between both models is similar except for two GPUs. On the RTX 2070, SYCL outperforms CUDA by 10% on average, while on the V100 the difference is still positive but slightly smaller (7%).

To find out more about the causes of these larger performance differences, we have profiled both code executions on the RTX 2070 and the RTX 3090 GPUs using the NVIDIA Nsight Compute tool [31].[21] Table 4 presents some relevant metrics collected from this experimental task. As can be seen, SYCL outperforms CUDA for several metrics on the RTX 2070, not only in memory management but also in computational productivity. However, both codes achieve practically the same values on

[20] While both SYCL and CUDA codes were executed, only the SYCL version is included to improve the readability of the chart.

[21] The profiling metrics used are described in https://docs.nvidia.com/nsight-compute/ProfilingGuide/.

Table 3 SLOC of SW# (CUDA and DPC++ versions)

CUDA[1]		DPC++[2]		
Original	Not migrated by SYCLomatic	SYCLomatic output	Hand-tuned	Pure SYCL
8072	408	10457	12175	9866

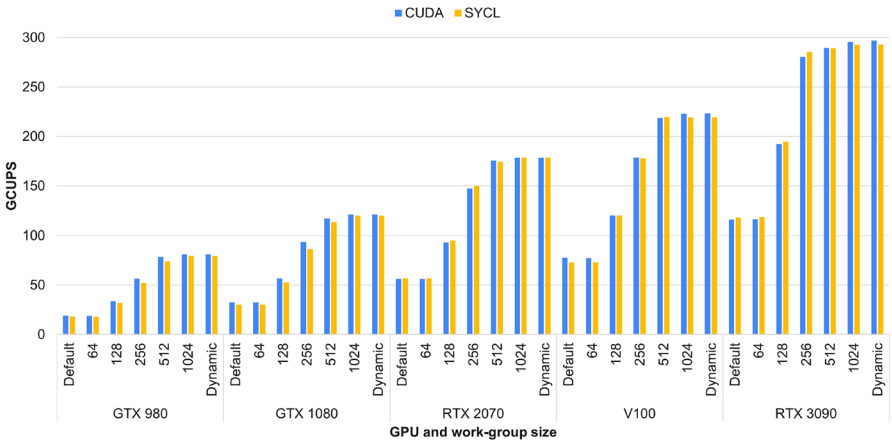

Fig. 2 Performance comparison when varying work-group size

the RTX 3090. At this point, we assume that it could be related to particular features of their micro-architecture in contrast to the rest of them.[22]

4.2.2 Cross-GPU-vendor and cross-architecture portability results

To verify cross-vendor GPU portability, the SYCL code was run on two AMD GPUs and two Intel GPUs for searching the Env. NR database, covering both discrete and integrated segments. Similarly, the same code was executed on four Intel CPUs and one AMD CPU to demonstrate its cross-architecture portability (see Fig. 8). In both cases, all results were verified to be correct. At this point, little can be said about their performance due to the absence of an optimized version for these devices. Finally, it is important to remark on two aspects: (1) running these tests just required a single back-end switch; (2) as the ported DPC++ version is pure SYCL code, running this version on different architectures just needs a compatible compiler. As it was mentioned before, several are available nowadays and this aspect is analyzed in the next Section.

[22] Both GPUs belong to the same Nvidia micro-architecture; unlike previous generations, Nvidia has employed different codenames for each commercial segment (Turing is the codename for the consumer segment while Volta is corresponding for the professional one).

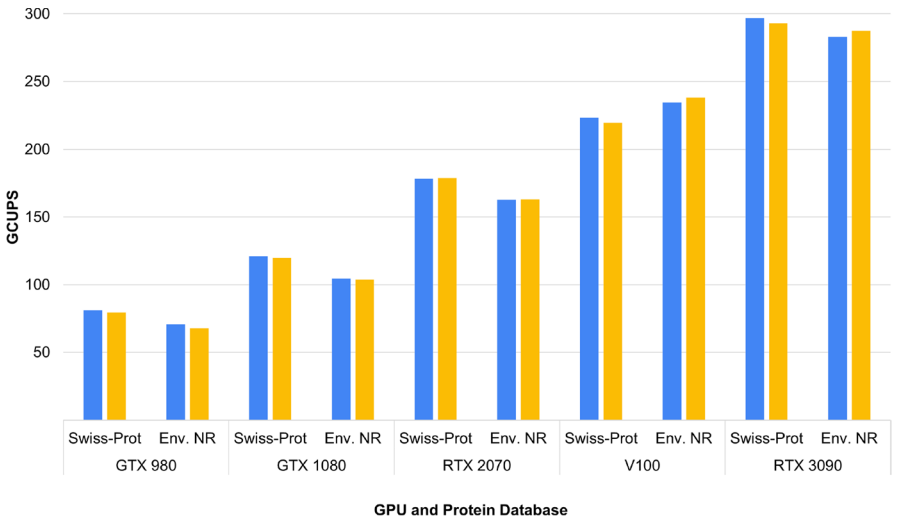

Fig. 3 Performance comparison when varying protein databases

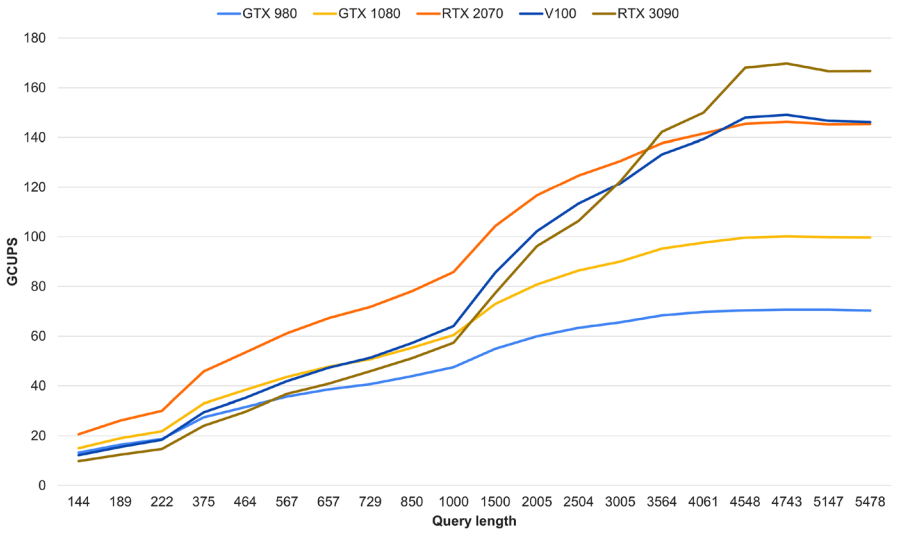

Fig. 4 Performance comparison when varying the query length

4.2.3 Cross-SYCL-implementation portability results

To verify cross-SYCL-implementation portability, we decided to compile and run the ported code on several GPUs and CPUs using the AdaptiveCpp framework. For this task, the SYCL standardization step from Sect. 3.1.6 was fundamental. Figure 9 presents the performance achieved for both oneAPI and AdaptiveCpp versions when searching the Swiss-Prot database. Some performance losses can be

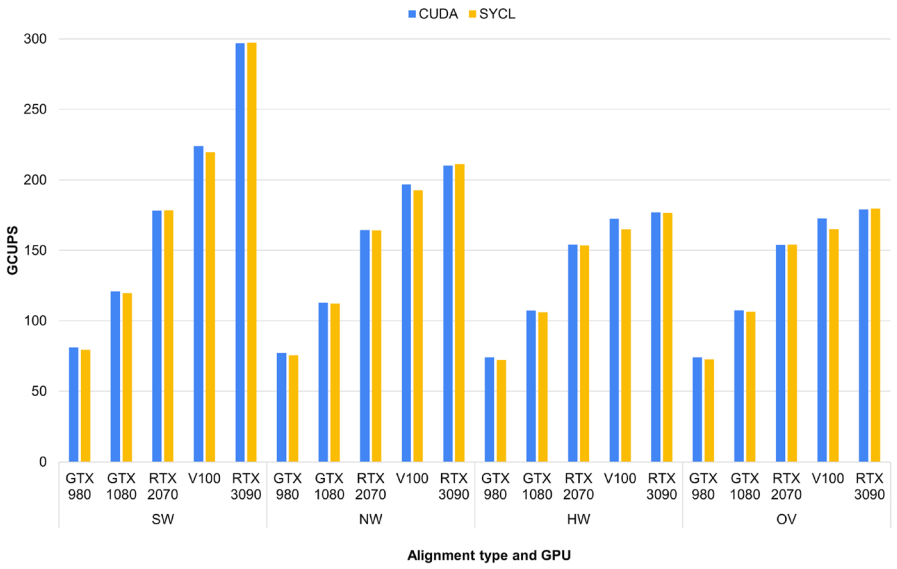

Fig. 5 Performance comparison when varying the alignment algorithm

observed in AdaptiveCpp when using the *generic* compiler instead of the *specific-target* one (up to 15%). In the opposite direction, no significant performance differences can be noted between oneAPI and AdaptiveCpp, except for the Arc A770

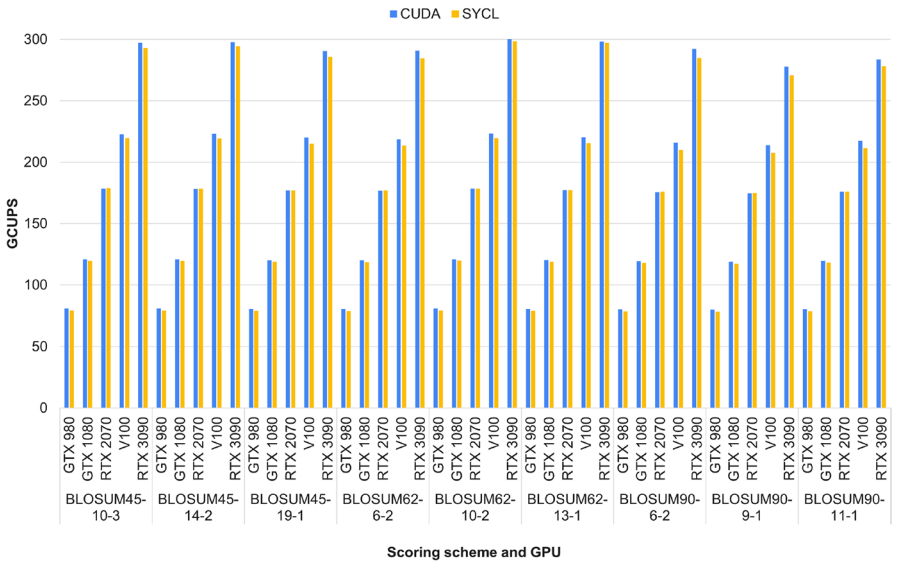

Fig. 6 Performance comparison when varying the scoring scheme

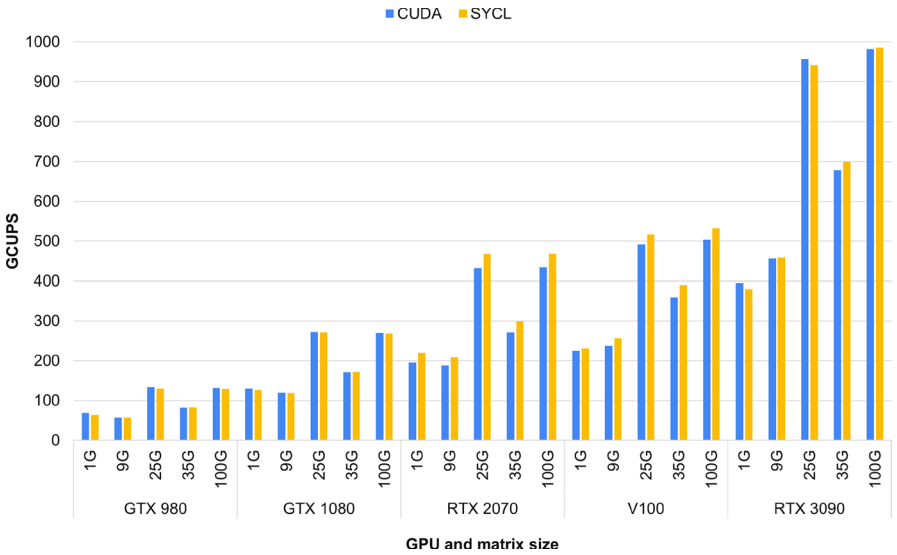

Fig. 7 Performance comparison for DNA alignment

Table 4 Profiles of DNA sequence alignment executions (matrix size: 100 G) for CUDA and SYCL on RTX 2070 and RTX 3090 GPUs

Section Name	Metric Name	RTX 2070			RTX 3090		
		CUDA	SYCL	SYCL/CUDA ratio	CUDA	SYCL	SYCL/CUDA ratio
Compute workload Analysis	Executed Ipc Active (inst/cycle)	2.15	2.71	1.26	2.18	2.07	0.95
	Executed Ipc Elapsed (inst/cycle)	1.79	2.26	1.26	1.53	1.46	0.95
GPU Speed Of Light Throughput	Memory Throughput (%)	20.08	33.85	1.69	N/A	N/A	–
	DRAM Throughput (%)	0.16	0.20	1.25	0.42	0.42	1
	L1/TEX Cache Throughput (%)	24.18	40.74	1.68	25.72	24.43	0.95
	L2 Cache Throughput (%)	0.17	0.24	1.41	0.35	0.3	0.95
Memory Workload Analysis	Memory Throughput (Mbyte/sec)	677.46	853.17	1.26	3684.16	3503.77	0.95
	Max Bandwidth (%)	20.08	33.85	1.69	18.16	17.26	0.95
	Mem Pipes Busy (%)	20.08	33.84	1.69	18.16	17.26	0.95
Scheduler statistics	Issued Warp Per Scheduler	0.54	0.68	1.26	N/A	N/A	–

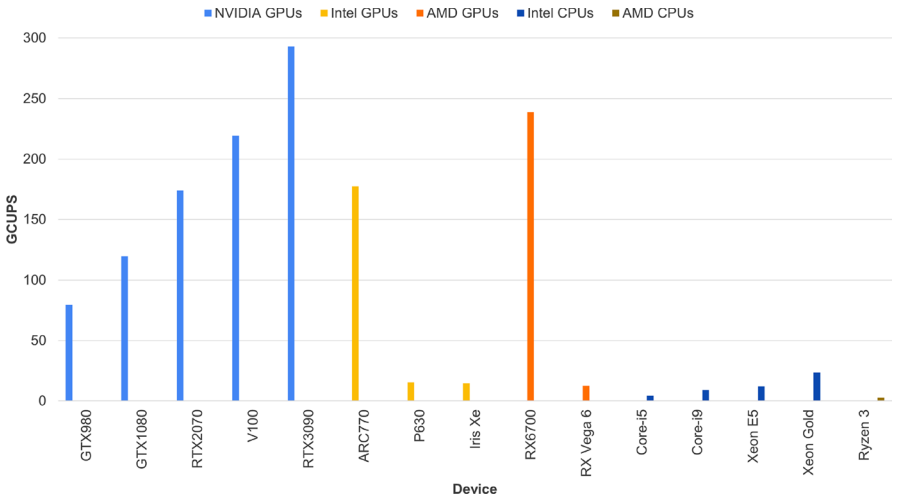

Fig. 8 Performance of SYCL code on different vendor GPUs and CPUs

GPU, where the former is just 1.05× faster than the latter. This fact contributes to the interchangeability of SYCL and favors its adoption.

4.3 Related works

Some preliminary studies assessing the portability of SYCL and oneAPI can be found in simulation [10], math [32, 33], machine learning [34, 35], software benchmarks [36, 37], image processing [38], and cryptography [39]. In the bioinformatics field, some works can also be mentioned. In [9], the authors describe the experience of translating a CUDA implementation of a high-order epistasis detection algorithm to SYCL, finding that the highest performance of both versions is comparable on an NVIDIA V100 GPU. It is important to remark that some hand-tuning was required in the SYCL implementation to reach its maximum performance. In [40], the authors migrate representative kernels in bioinformatics applications from CUDA to SYCL and evaluate their performance on an NVIDIA V100 GPU, explaining the performance gaps through code profiling and analyzes. The performance difference ranges from 1.25× to 5× (2.73× on average) and the authors relate it to CUDA's mature and extensive development environment. As in the previous work, the authors did not report if manual or automatic migration was employed. In [41], the authors evaluate the performance and portability of the CUDA-based ADEPT kernel for SW short-read alignment. Unlike this study, the authors followed manual porting to obtain a DPC++ equivalent version of ADEPT, arguing that the resultant code was unnecessarily complex and required major changes. Both CUDA and DPC++ versions were run on an NVIDIA V100 GPU, where the latter was approximately 2× slower in all experiments. However, the authors were not able to determine the causes of the slowdown due to some limitations in kernel profiling. In addition, the code portability of the DPC++ version was verified on an Intel P630

GPU. In [42], the authors translate the molecular docking software AutoDock-GPU from CUDA to SYCL by employing `dpct`, remarking that this tool greatly reduces the effort of code migration but manual steps for code completion and tuning are still required. From a performance point of view, most test cases show that SYCL executions are slower than CUDA ones on an NVIDIA A100 (1.91× on average). While still preliminary analysis, the authors attribute performance gaps to the SYCL version performing more computations than its CUDA counterpart and higher register pressure and shared memory usage from the former. In [43], the authors presented *OneJoin*, a oneAPI-based tool to edit similarity join in DNA data decoding. This tool was developed using oneAPI from scratch and its portability was checked on two Intel CPUs (Xeon E-2176 G, Core i9-10920X), an integrated Intel GPU (P630), and a discrete NVIDIA GPU (RTX 2080). Last, few works have compared performance and portability of different SYCL implementations [44–46]. Generally, the performance results have been similar between SYCL implementations, but significant differences occurred in some cases. Even though, these results cannot be considered definitive due to the rapid growth of these tools.

In this work, as was shown above, *SW#* has been completely migrated with a small programmer intervention in terms of hand-coding. Moreover, it has been possible to port the migrated code between different GPU and CPU architectures from multiple vendors. Specifically, the code portability was verified on 5 NVIDIA GPU microarchitectures, 3 Intel GPU microarchitectures—one discrete and two integrated—2 AMD GPU microarchitecture, 1 AMD CPU microarchitecture, and 4 Intel CPU microarchitectures; with no noticeable performance degradation on the 5 different NVIDIA GPUs. In the same line, the performance remained stable when cross-SYCL-implementation portability was verified on 5 GPUs and 2 CPUs from different manufacturers.

5 Conclusions and future work

SYCL aims to take advantage of the benefits of hardware specialization while increasing productivity and portability at the same time. Recently, Intel released oneAPI, a complete programming ecosystem that follows the SYCL standard. In this paper, we have presented our experiences migrating a CUDA-based biological software to SYCL. Besides, the portability of the migrated code was analyzed in combination with a performance evaluation on different GPU and CPU architectures. The main findings of this research are:

- `SYCLomatic` has proved to be an efficient tool for migrating 95% of the original CUDA code to DPC++, according to Intel's marketing rates. A small hand-tune effort was required first to achieve a functional version and then a fully SYCL-compliant one.
- To minimize possible biases, the SYCL code was successfully executed on 5 NVIDIA GPUs from different microarchitectures (Maxwell, Pascal, Volta, Turing, and Ampere), 3 Intel GPUs (one integrated and two discrete), 2 AMD GPUs (one integrated and one discrete), 1 AMD CPU, and 4 different Intel CPUs. Extending and diversifying the set of experimental platforms reinforce

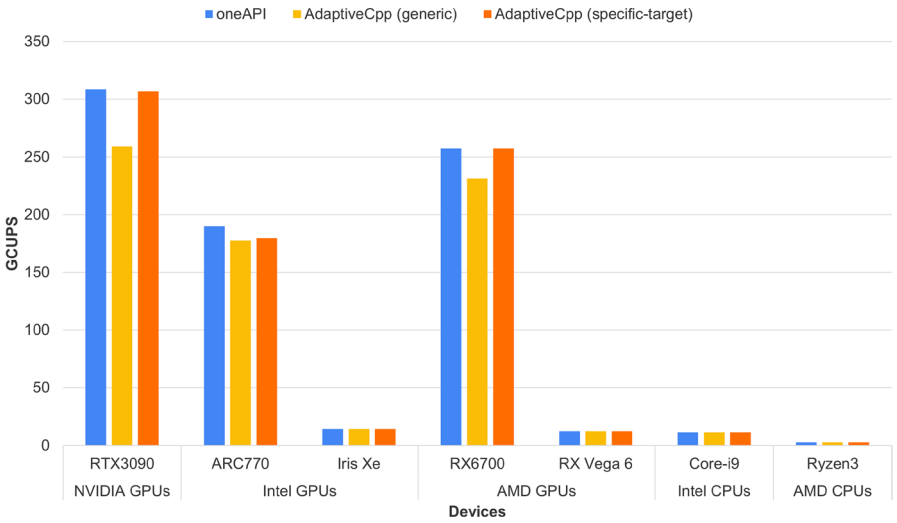

Fig. 9 Performance comparison between SYCL implementations (oneAPI and AdaptiveCpp) on different vendor GPUs and CPUs

the conclusions reached regarding cross-vendor GPU and cross-architecture portability of SYCL.

- Unlike our previous work, tests carried out included a wide variety of scenarios (sequence type, sequence size, alignment algorithm, and scoring scheme, among others) to represent diverse workloads in the field. This fact strengthens previous findings stating that performance results showed that both CUDA and SYCL versions presented comparable GCUPS, demonstrating that portability can be gained without severe performance losses.
- Last but not least, the portability between SYCL implementations was also verified, showing that performance remains stable when switching between oneAPI and AdaptiveCpp. This is another fact that favors the adoption of SYCL.

Given the results obtained, SYCL and its implementations can offer attractive opportunities for the bioinformatics community, especially considering the vast existence of CUDA-based legacy codes. In this regard, because SYCL is still under development, the advance of its compilers and the growth of the programmers' community will be key aspects in determining SYCL's success in improving productivity and portability.

Future work will focus on:

- Optimizing the SYCL code to reach its maximum performance. In particular, the original *SW#* suite does not consider some known optimizations for SW align-

ment [47], such as instruction reordering to reduce their count and the use of lower precision integers to increase parallelism[23].

- Running the SYCL code on other architectures such as FPGAs, to extend the cross-architecture portability study. In the same vein, considering hybrid CPU-GPU execution, taking advantage of the inherent ability of SYCL to exploit co-execution [48, 49].
- Carrying out an extensive study of the performance portability of these codes, following Marowka's proposal [50].

Acknowledgements Not applicable.

Author Contributions ER and CG-S proposed the idea. MC developed the software and conducted the experiments. ER, MC, and CG-S analyzed the results and wrote the paper. MN and MP-M reviewed the manuscript. All authors contributed to the article and approved the submitted version.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. Grant PID2021-126576NB-I00 funded by MCIN/AEI/10.13039/501100011033 and, as appropriate, by "ERDF A way of making Europe", by the "European Union" or by the "European Union Next Generation EU/PRTR".

Data availability statements The SW# CUDA software used in this study is available at https://github.com/mkorpar/swsharp. The migration to SYCL was performed using the SYCLomatic tool, accessible at https://github.com/oneapi-src/SYCLomatic. The migrated SW# software is available at https://github.com/ManuelCostanzo/swsharp_sycl. The protein data utilized for this research are sourced from the UniProtKB/Swiss-Prot (Swiss-Prot) database (release 2022_07), which can be found at https://www.uniprot.org/downloads, and the Environmental Non-Redundant (Env. NR) database (release 2021_04) available at https://ftp.ncbi.nlm.nih.gov/blast/db/. Operating system: Platform independent. Programming languages: C++20, CUDA 11.7. Other requirements: Intel LLVM available at https://github.com/intel/llvm with the CUDA toolchain available at https://intel.github.io/llvm-docs/GetStartedGuide.html#build-dpc-toolchain-with-support-for-nvidia-cuda. AdaptiveCpp es available athttps://github.com/AdaptiveCpp/AdaptiveCpp.

Declarations

Conflict of interest The authors declare that they have no competing interests.

Ethics approval and consent to participate Not applicable.

Consent for publication Not applicable.

[23] It is important to note that at the time of *SW#*'s development, most CUDA-enabled GPUs did not support efficient arithmetic on 8-bit vector data types.

References

1. Dally WJ, Turakhia Y, Han S (2020) Domain-specific hardware accelerators. Commun ACM 63(7):48–57. https://doi.org/10.1145/3361682
2. Robert D (2021) GPU shipments increase year-over-year in Q3. https://www.jonpeddie.com/press-releases/gpu-shipments-increase-year-over-year-in-q3
3. Nobile MS, Cazzaniga P, Tangherloni A, Besozzi D (2016) Graphics processing units in bioinformatics, computational biology and systems biology. Brief Bioinform 18(5):870–885. https://doi.org/10.1093/bib/bbw058
4. De Oilveira Sandes EF, Boukerche A, De Melo ACMA (2016) Parallel optimal pairwise biological sequence comparison: algorithms, platforms, and classification. ACM Comput Surv. https://doi.org/10.1145/2893488
5. Ohue M, Shimoda T, Suzuki S, Matsuzaki Y, Ishida T, Akiyama Y (2014) Megadock 4.0: an ultra-high-performance protein-protein docking software for heterogeneous supercomputers. Bioinformatics 30(22):3281–3283
6. Loukatou S, Papageorgiou L, Fakourelis P, Filntisi A, Polychronidou E, Bassis I, Megalooikonomou V, Makałowski W, Vlachakis D, Kossida S (2014) Molecular dynamics simulations through GPU video games technologies. J Mole Biochem 3(2):64
7. Mrozek D, Brożek M, Małysiak-Mrozek B (2014) Parallel implementation of 3d protein structure similarity searches using a GPU and the CUDA. J Mol Model 20(2):1–17
8. Group K (2009) The OpenCL specification. Version 1.0. https://www.khronos.org/registry/cl/specs/opencl-1.0.pdf
9. Jin Z, Vetter JS (2022) Performance portability study of epistasis detection using sycl on nvidia gpu. In: Proceedings of the 13th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics. BCB '22. Association for Computing Machinery, New York. https://doi.org/10.1145/3535508.3545591
10. Christgau S, Steinke T (2020) Porting a legacy CUDA stencil code to oneAPI. In: 2020 IEEE IPDPSW, pp 359–367. https://doi.org/10.1109/IPDPSW50202.2020.00070
11. Korpar M, Sikic M (2013) SW# - GPU-enabled exact alignments on genome scale. Bioinformatics 29(19):2494–2495. https://doi.org/10.1093/bioinformatics/btt410
12. Costanzo M, Rucci E, García-Sánchez C, Naiouf M, Prieto-Matías M (2022) Migrating CUDA to oneAPI: a smith-waterman case study. In: Rojas I, Valenzuela O, Rojas F, Herrera LJ, Ortuño F (eds) Bioinform Biomed Eng. Springer, Cham, pp 103–116
13. De O, Sandes EF, Miranda G, Martorell X, Ayguade E, Teodoro G, De Melo ACMA (2016) Masa: a multiplatform architecture for sequence aligners with block pruning. ACM Trans Parallel Comput 2(4):28–12831. https://doi.org/10.1145/2858656
14. Needleman SB, Wunsch CD (1970) A general method applicable to the search for similarities in the amino acid sequence of two proteins. J Mol Biol 48(3):443–453. https://doi.org/10.1016/0022-2836(70)90057-4
15. Smith TF, Waterman MS (1981) Identification of common molecular subsequences. J Mol Biol 147(1):195–197
16. Hasan L, Al-Ars Z (2011) In: Lopes H, Cruz L (eds) An overview of hardware-based acceleration of biological sequence alignment, pp 187–202. Intech
17. Isaev A (2006) Introduction to mathematical methods in bioinformatics. Universitext, 1st edn. Springer, Heidelberg
18. Daily J (2016) Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. BMC Bioinform. https://doi.org/10.1186/s12859-016-0930-z
19. Mneimneh S (2024) Computational biology lecture 4: overlap detection, Local Alignment, Space Efficient Needleman–Wunsch
20. Korpar M, Sosic M, Blazeka D, Sikic M (2016) SWdb: GPU-accelerated exact sequence similarity database search. PLoS ONE 10(12):1–11. https://doi.org/10.1371/journal.pone.0145857
21. Khoo AA, Ogrizek-Tomaš M, Bulović A, Korpar M, Gürler E, Slijepčević I, Šikić M, Mihalek I (2013) ExoLocator-an online view into genetic makeup of vertebrate proteins. Nucl Acids Res 42(D1):879–881. https://doi.org/10.1093/nar/gkt1164
22. Ghorpade J, Parande J, Kulkarni M, Bawaskar A (2012) Gpgpu processing in CUDA architecture. arXiv:1202.4347

23. Software (2023) ComputeCpp Comunity Edition. https://developer.codeplay.com/products/computecpp/ce/home
24. Intel Corp (2021) Intel oneAPI. https://software.intel.com/en-us/oneapi
25. The triSYCL project. https://github.com/triSYCL/triSYCL (2023)
26. Alpay: OpenSYCL implementation. https://github.com/AdaptiveCpp/AdaptiveCpp (2023)
27. Alpay A, Soproni B, Wünsche H, Heuveline V (2022) Exploring the possibility of a hipsycl-based implementation of oneapi. In: International workshop on OpenCL. IWOCL'22. Association for Computing Machinery, New York. https://doi.org/10.1145/3529538.3530005
28. Alpay A, Heuveline V (2023) One pass to bind them: The first single-pass sycl compiler with unified code representation across backends. In: Proceedings of the 2023 international workshop on OpenCL. IWOCL '23. Association for Computing Machinery, New York. https://doi.org/10.1145/3585341.3585351
29. Rucci E, Garcia C, Botella G, Giusti AED, Naiouf M, Prieto-Matias M (2018) Oswald: Opencl smith-waterman on altera's FPGA for large protein databases. Int J High Perform Comput Appl 32(3):337–350. https://doi.org/10.1177/1094342016654215
30. Rucci E, Garcia C, Botella G, De Giusti A, Naiouf M, Prieto-Matias M (2018) SWIFOLD: Smith-waterman implementation on FPGA with OpenCL for long DNA sequences. BMC Syst Biol 12(Suppl 5):96. https://doi.org/10.1186/s12918-018-0614-6
31. NVIDIA (2022) Nsight Compute. https://developer.nvidia.com/nsight-compute
32. Tsai YM, Cojean T, Anzt H (2021) Porting a sparse linear algebra math library to Intel GPUs
33. Costanzo M, Rucci E, Sanchez CG, Naiouf M (2021) Early experiences migrating cuda codes to oneapi. In: Short Papers of the 9th Conference on Cloud Computing Conference, Big Data and Emerging Topics, pp 14–18. http://sedici.unlp.edu.ar/handle/10915/125138
34. Martínez PA, Peccerillo B, Bartolini S, García JM, Bernabé G (2022) Applying intel's oneAPI to a machine learning case study. Concurrency Comput Pract Exp 34(13):6917. https://doi.org/10.1002/cpe.6917
35. Faqir-Rhazoui Y, García C (2023) Exploring the performance and portability of the k-means algorithm on SYCL across CPU and GPU architectures. J Supercomput 79(16):18480–18506. https://doi.org/10.1007/s11227-023-05373-2
36. Jin Z, Vetter J (2021) Evaluating cuda portability with HIPCL and DPCT. In: 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp 371–376. https://doi.org/10.1109/IPDPSW52791.2021.00065
37. Castaño G, Faqir-Rhazoui Y, García C, Prieto-Matías M (2022) Evaluation of intel's DPC++ compatibility tool in heterogeneous computing. J Parall Distrib Comput 165:120–129. https://doi.org/10.1016/j.jpdc.2022.03.017
38. Yong W, Yongfa Z, Scott W, Wang Y, Qing X, Chen W (2021) Developing medical ultrasound imaging application across gpu, fpga, and CPU using oneapi. In: International workshop on OpenCL. IWOCL'21. Association for Computing Machinery, New York. https://doi.org/10.1145/3456669.3456680
39. Marinelli E, Appuswamy R (2021) Xjoin: portable, parallel hash join across diverse xpu architectures with OneaPI. In: Proceedings of the 17th international workshop on data management on new hardware. DAMON '21. Association for Computing Machinery, New York. https://doi.org/10.1145/3465998.3466012
40. Jin Z, Vetter JS (2022) Understanding performance portability of bioinformatics applications in sycl on an nvidia gpu. In: 2022 IEEE International Conference on Bioinformatics and Biomedicine (BIBM), pp 2190–2195. https://doi.org/10.1109/BIBM55620.2022.9995222
41. Haseeb M, Ding N, Deslippe J, Awan M (2021) Evaluating performance and portability of a core bioinformatics kernel on multiple vendor GPUS. In: 2021 International workshop on performance, portability and productivity in HPC (P3HPC), pp 68–78. https://doi.org/10.1109/P3HPC54578.2021.00010
42. Solis-Vasquez L, Mascarenhas E, Koch A (2023) Experiences migrating cuda to sycl: a molecular docking case study. In: Proceedings of the 2023 international workshop on OpenCL. IWOCL '23. Association for Computing Machinery, New York. https://doi.org/10.1145/3585341.3585372
43. Marinelli E, Appuswamy R (2021) OneJoin: cross-architecture, scalable edit similarity join for DNA data storage using oneAPI. In: ACM (ed) ADMS 2021, 12th international workshop on accelerating analytics and data management systems using modern processor and storage architectures, in conjunction with VLDB 2021, 16 August 2021, Copenhagen, Denmark, Copenhagen

44. Johnston B, Vetter JS, Milthorpe J (2020) 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. https://doi.org/10.1109/P3HPC51967.2020.00010
45. Breyer M, Daiß G, Pflüger D (2021) Performance-portable distributed k-nearest neighbors using locality-sensitive hashing and sycl. In: International workshop on OpenCL. IWOCL'21. Association for Computing Machinery, New York. https://doi.org/10.1145/3456669.3456692
46. Shilpage WR, Wright SA (2023) An investigation into the performance and portability of sycl compiler implementations. In: Bienz A, Weiland M, Baboulin M, Kruse C (eds) High performance computing. Springer, Cham, pp 605–619
47. Rognes T (2011) Faster Smith–Waterman database searches with inter-sequence SIMD parallelization. BMC Bioinform 12:221
48. Constantinescu D-A, Navarro A, Corbera F, Fernández-Madrigal J-A, Asenjo R (2021) Efficiency and productivity for decision making on low-power heterogeneous cpu+gpu socs. J Supercomput 77(1):44–65. https://doi.org/10.1007/s11227-020-03257-3
49. Nozal R, Bosque JL (2021) Exploiting co-execution with OneAPI: heterogeneity from a modern perspective. In: Sousa L, Roma N, Tomás P (eds) Euro-Par 2021: parallel processing. Springer, Cham, pp 501–516
50. Marowka A (2022) Reformulation of the performance portability metric. Softw Pract Exp 52(1):154–171. https://doi.org/10.1002/spe.3002

Authors and Affiliations

Manuel Costanzo[1] · Enzo Rucci[1] · Carlos García-Sanchez[2] · Marcelo Naiouf[1] · Manuel Prieto-Matías[2]

✉ Carlos García-Sanchez
garsanca@ucm.es

Manuel Costanzo
mcostanzo@lidi.info.unlp.edu.ar

Enzo Rucci
erucci@lidi.info.unlp.edu.ar

Marcelo Naiouf
mnaiouf@lidi.info.unlp.edu.ar

Manuel Prieto-Matías
mpmatias@ucm.es

1 Facultad de Informática, III-LIDI, UNLP – CIC, 1900 La Plata, Buenos Aires, Argentina

2 Facultad Informatica. DACyA, Universidad Complutense de Madrid, 28040 Madrid, Spain