

Comparing Performance and Portability between CUDA and SYCL for Protein Database Search on NVIDIA, AMD, and Intel GPUs

Manuel Costanzo
III-LIDI, Facultad de Informática, UNLP - CIC
La Plata, Argentina
0000-0002-6937-3943

Enzo Rucci
III-LIDI, Facultad de Informática, UNLP - CIC
La Plata, Argentina
0000-0001-6736-7358

Carlos García-Sánchez
Dpto. Arquitectura de Computadores y Automática, Universidad Complutense de Madrid
Madrid, España
0000-0002-3470-1097

Marcelo Naiouf
III-LIDI, Facultad de Informática, UNLP - CIC
La Plata, Argentina
0000-0001-9127-3212

Manuel Prieto-Matías
Dpto. Arquitectura de Computadores y Automática, Universidad Complutense de Madrid
Madrid, España
0000-0003-0687-3737

Abstract—The heterogeneous computing paradigm has led to the need for portable and efficient programming solutions that can leverage the capabilities of various hardware devices, such as NVIDIA, Intel, and AMD GPUs. This study evaluates the portability and performance of the SYCL and CUDA languages for one fundamental bioinformatics application (Smith-Waterman protein database search) across different GPU architectures, considering single and multi-GPU configurations from different vendors. The experimental work showed that, while both CUDA and SYCL versions achieve similar performance on NVIDIA devices, the latter demonstrated remarkable code portability to other GPU architectures, such as AMD and Intel. Furthermore, the architectural efficiency rates achieved on these devices were superior in 3 of the 4 cases tested. This brief study highlights the potential of SYCL as a viable solution for achieving both performance and portability in the heterogeneous computing ecosystem.

Index Terms—oneAPI, SYCL, GPU, CUDA, Performance portability

I. INTRODUCTION

In the last decade, the quest to improve the energy efficiency of computing systems has fueled the trend toward heterogeneous computing and massively parallel architectures [1]. Nowadays, GPUs can be considered the dominant accelerator, and Nvidia, Intel, and AMD are the most prominent manufacturers. In the 4th quarter of 2022, Intel and AMD had 9% of the market, with Nvidia dominating the discrete graphics card market at 82%. Moreover, considering also the integrated and embedded graphics, Intel had 71% quote, Nvidia 17%, and AMD 12%¹. This poses a significant challenge for researchers who use GPUs for their experiments and simulations. The critical question is how to use this growing computational

capacity transparently without having to pay attention to the programming models, hardware support, or mandatory software ecosystem.

Focusing on the programming aspect, CUDA is still the most popular programming language for GPUs, although it is a proprietary language only valid for NVIDIA devices. Fortunately, other open initiatives have contemplated the programming of GPUs or even other accelerators generically. In particular, SYCL is one of the most promising recent initiatives, which is an open standard from the Khronos Group. One noteworthy feature of SYCL is its status as a cross-platform abstraction layer, enabling programmers to adhere to the fundamental principle of “write code once and run it anywhere”. In this sense, the same SYCL code can run not only on multiple vendor GPUs but also on different hardware platforms, including CPUs and FPGAs. SYCL capitalizes on programming productivity by reducing the effort required during development tasks and minimizing maintenance costs. The concept of *performance portability* becomes fundamental in this context. Specifically, performance portability encompasses two key aspects: (1) enabling the execution of a single application on various hardware platforms, and (2) achieving a desired level of performance across these diverse platforms [2].

This paper aims to address the previous issue by exploring the SYCL programming paradigm in the field of Bioinformatics and Computational Biology. These research areas have been leveraging GPUs for over two decades [3] and numerous of their implementations are based on CUDA, imposing significant limitations on portability across a wide range of heterogeneous architectures. For that reason, this study evaluates the portability and performance of the SYCL and CUDA languages for one fundamental bioinformatics application (Smith-Waterman biological sequence alignment)

¹<https://www.pcgamer.com/intel-is-already-matching-amd-for-gaming-graphics-market-share/>

across different GPU architectures, considering single and multi-GPU configurations from multiple vendors. Hence, we select the *SW#* suite [4], [5]: a CUDA-based, memory-efficient implementation for biological sequence alignment, that has been recently migrated to SYCL [6]. Our main contributions can be summarized as:

- An adaptation and extension of the performance model from [7]. This performance model is adapted to the features of the *SW#* suite and also extended to include AMD and Intel GPUs (both discrete and integrated types).
- A functional and performance portability study for *SW#* applications across different GPU architectures, considering single and multi-GPU configurations from multiple vendors. To the best of our knowledge, no previous study has considered such a diverse and large set of GPUs.

The rest of the paper is organized as follows. Section II introduces the background for this research. Section III describes the case-study applications and also the adapted and extended performance model. Section IV presents the functional and performance portability results. Finally, Section V discusses some related works, and Section VI presents the conclusions and possible lines for future work.

II. BACKGROUND

A. GPUs and Programming Models

In 2007, Nvidia introduced CUDA [8] alongside the Tesla GPU, to enable general-purpose programming on GPUs. CUDA is a programming model and parallel computing platform specifically designed for general computing on GPUs. While CUDA has become the most popular low-level programming model for general-purpose GPU computing, its main limitation is that it only supports NVIDIA devices. In the opposite sense, OpenCL [9] gained prominence because it can be used in several devices and vendors requiring a similar abstraction level as CUDA.

High-Level Programming initiatives such as OpenMP [10], OpenACC [11], [12], and SYCL [13] have played significant roles in the field of parallel computing in GPU scenarios. OpenMP initially focused on multi-core CPU computing but later expanded its support to include accelerators like GPUs with the release of v4.0. While OpenACC [14] (Open Accelerators) emerged as one of the earliest high-level approaches for GPU programming through the use of directive-based programming, OpenMP has even started overshadowing it by incorporating most of their features.

Currently, one of the most promising initiatives in the GPU programming ecosystem is SYCL [13]. It enables developers to write code for heterogeneous processors using standard ISO C++. It incorporates host and kernel code in a single source file and utilizes templates and lambda functions for generic programming. Moreover, SYCL supports various acceleration APIs, such as OpenCL, enabling seamless integration with lower-level code.

Multiple SYCL implementations are available nowadays: Codeplay's ComputeCpp [15], oneAPI by Intel [16],

triSYCL [17] led by Xilinx, and OpenSYCL [18] (previously denoted as hipSYCL) led by Heidelberg University. In particular, Intel oneAPI can be considered the most mature developer suite. Among the main features of oneAPI, we can find that is an open, cross-industry project that aims to provide an efficient, high-performance programming model. It eliminates the concept of separate code bases for host and device such as in OpenCL. Moreover, multiple programming languages and different tools for each architecture are supported. Data Parallel C++ (DPC++) is oneAPI's core language for programming accelerators and multiprocessors [16], which integrates SYCL and OpenCL standards without additional extensions. Additionally, oneAPI facilitates interoperability with optimized libraries such as oneCCL, oneDAL, oneDNN, oneMKL, oneTBB, and oneVPL, catering to diverse parallel application domains.

B. Smith-Waterman Algorithm

The SW algorithm is widely used to obtain the optimal local alignment between two sequences [19]. This method is based on a dynamic programming approach and is highly sensitive since it explores all possible alignments between the sequences.

Given two sequences Q and D of length $|Q| = m$ and $|D| = n$, the recurrence relations for the SW algorithm with the modification of Gotoh [20] are defined as follows:

$$H_{i,j} = \max \begin{cases} 0 \\ H_{i-1,j-1} + SM(Q[i], D[j]) \\ E_{i,j} \\ F_{i,j} \end{cases} \quad (1)$$

$$E_{i,j} = \max \begin{cases} H_{i,j-1} - G_o \\ E_{i,j-1} - G_e \end{cases} \quad (2)$$

$$F_{i,j} = \max \begin{cases} H_{i-1,j} - G_o \\ F_{i-1,j} - G_e \end{cases} \quad (3)$$

The similarity score $H_{i,j}$ is computed to identify a common subsequence; $H_{i,j}$ contains the score for aligning the prefixes $Q[1..i]$ and $D[1..j]$. Moreover, $E_{i,j}$ and $F_{i,j}$ correspond to the scores of prefix $Q[1..i]$ and $D[1..j]$ aligned to a gap, respectively. SM denotes the *scoring matrix* and defines the match/mismatch scores between residues. Last, G_o and G_e refer to the gap open and gap extension penalties, respectively.

First of all, H , E and F must be initialized with 0 when $i = 0$ or $j = 0$. Then, the recurrences should be calculated with $1 \leq i \leq m$ and $1 \leq j \leq n$. The highest value in the H matrix (S) corresponds to the optimal local alignment score between $Q[1..i]$ and $D[1..j]$. If required, the optimal local alignment is finally obtained by following a traceback procedure whose starting point is S . From a computational point of view, it is important to highlight the computational dependencies of any H element. Any cell can be calculated only after the values of the upper, left, and upper-left neighbors are known; imposing restrictions on how H can be processed.

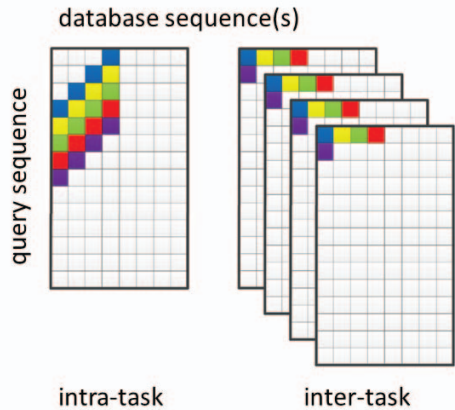


Fig. 1. Parallelization approaches in similarity matrix computations (adapted from [22]). Each color indicates the cells that can be computed together in a SIMD manner.

SW in practice and parallelization issues. The SW algorithm can be used to compute: (a) pairwise alignments (one-to-one); usually associated with long DNA sequences; or (b) database similarity searches (one-to-many), usually associated with protein sequence alignment. Although the processing nature of the SW algorithm with the data dependencies on the computation $H_{i,j}$ is very challenging from the point of view of parallelism exploitation, both approaches have been studied in the literature exploiting the SIMD capabilities. In the (a) case, 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 the (b) approach, while the intra-task scheme can be used, a better parallel scheme consists in simultaneously calculating multiple matrices without communication between the PEs (*inter-task parallelism*) [21] could be performed. Fig. 1 illustrates both approaches.

The SW algorithm runs in quadratic time and space to compute optimal alignment. However, computing optimal alignment scores do not require storing the full similarity matrix and can be calculated in linear space complexity. Similarity database search takes advantage of this feature since optimal alignment only makes sense for very similar sequences. Therefore, all alignment scores are calculated first and optimal alignments are computed only for top-ranked database sequences.

C. Performance portability

According to Penycook et al. [23], *performance portability* refers to “A measurement of an application’s performance efficiency for a given problem that can be executed correctly on all platforms in a given set”. These authors define two different performance efficiency metrics: *architectural efficiency* and *application efficiency*. The former denotes the capacity of an application to effectively utilize hardware resources, measured as a proportion of the theoretical peak performance. The latter signifies the application’s ability to select the most suitable

implementation for each platform, representing a fraction of the highest observed performance achieved.

The metric for performance portability presented by Penycook et al. [23] was later reformulated by Marowka [2] to address some of its flaws. Formally, for a given set of platforms H from the same architecture class, the performance portability $\bar{\Phi}$ of a case-study application α solving problem p is:

$$\bar{\Phi}(\alpha, p, H) = \begin{cases} \frac{\sum_{i \in H} e_i(\alpha, p)}{|H|} & \text{if } i \text{ is supported } \forall i \in H \\ \text{not applicable (NA)} & \text{otherwise} \end{cases}$$

where $e_i(\alpha, p)$ corresponds to the performance efficiency of case-study application α solving problem p on the platform i .

The *performance portability* concept emphasizes the capability to write code that can efficiently utilize the available computing resources, such as CPUs, GPUs, or specialized accelerators while maintaining high performance regardless of the specific hardware configuration. With performance portability, developers can write code once and have it deliver optimal performance on various target platforms. This eliminates the need for extensive manual code optimizations or platform-specific modifications, reducing development time and effort.

III. CASE-STUDY APPLICATIONS AND PERFORMANCE MODEL

A. Case-Study Applications

Two GPU-accelerated implementations of p =protein database search were considered for the performance portability evaluation:

- **CUDA:** this version corresponds to the *SW#* suite, a CUDA-based, memory-efficient implementation for biological sequence alignment, which can be used either as a stand-alone application or a library. It can compute pairwise alignments as well as database similarity searches, for both protein and DNA sequences; and it allows configuring the alignment method (including SW), the open/extension penalties, and the scoring matrix. *SW#* combines CPU and GPU computation for optimal efficiency. It dynamically balances the workload between the CPU and GPU based on sequence lengths, aiming to minimize idle threads. From a parallelization point of view, *SW#* uses both inter-task and intra-task parallelism but primarily on the GPU side. The GPU divides the workload into two partitions: a “short kernel” process shortest database sequences using inter-task scheme, while a “long kernel” aligns longest sequences by intra-task strategy. When utilizing multiple GPUs, *SW#* follows a flexible approach: if the number of query sequences to be aligned is fewer than the number of available GPU devices, all devices align the same query sequence with a different database partition in synchronized manner. Conversely, if the number of query sequences is greater than the number of GPUs, each GPU align a different one against the complete database [4], [5].

- SYCL: this code is based on the implementation presented in the paper [6], representing a SYCL equivalent. The migration of the *SW#* suite was performed using `dpct` (the Data Parallel Compatibility Tool available in the oneAPI suite) and some hand-coding modifications.

B. Performance Model

Peak theoretical hardware performance must be estimated for all selected GPUs in this study to compute the performance portability metric. This step requires considering both hardware and algorithm features. Fortunately, the previous work from Lan et al. [7] can be used as a basis for this task; in this paper, the computing capability of different devices (including accelerators based on NVIDIA GPUs, Intel CPUs, and the discontinued Intel Xeon Phis) can be estimated using Eq. 4:

$$Capability = Clock_Rate \times Throughput \times Lanes \quad (4)$$

where *Clock_rate* refers to the clock frequency, *Throughput* refers to the instruction count that the device can execute in one clock cycle, and *Lanes* refers to the number of SIMD vector lanes. Then, the number of instructions issued in each cell update of the similarity matrix should be counted. In the sequence alignment context, the most popular metric for measuring performance is related to the number of Cell Updated Per Second (CUPS). So the theoretical peak performance of any device could be modeled using Eq. 5:

$$Theo_peak = \frac{Capability}{Instruction_count_one_cell_update} \quad (5)$$

Even though this study only considers GPUs, these equations can serve as a basis to estimate their theoretical peak performance for other devices such as CPUs or FPGAs. For this work, the previous performance model from [7] is adapted to the features of the *SW#* algorithm and also extended to other GPUs vendors such as AMD and Intel GPUs (both discrete and integrated types). Table I summarized the theoretical peak performance of selected GPUs using the Eq. 5. More details can be found in the rest of this section.

1) *SW# core instructions*: *SW#* computes the similarity matrix using 32-bit integers and performs 12 instructions per cell update. Algorithm 1 presents the snippet of cell update in similarity matrix as in Eq. 1, Eq. 2, and Eq. 3. Just adding, subtracting, and maximum instructions are required to perform a single-cell update.

2) *Architectural features on NVIDIA's GPU*: The # Cores in an NVIDIA GPU refers to the number of Streaming Multiprocessors. CUDA does not strictly follow a SIMD execution model but it adopts a similar one denoted as the SIMT model. A *warp* is composed of a group of 32 threads that execute the same instruction stream. According to [7], "a *warp* in SIMT is equivalent to a *vector* in SIMD, and a *thread* in SIMT is equivalent to a *vector lane* in SIMD". The instruction

Algorithm 1 Core instructions per cell update in similarity matrix

- 1: $E^1 = E_l - G_e$ ▷ E_l : E of its left neighbor
 - 2: $E^2 = H_l - G_o$ ▷ H_l : H of its left neighbor
 - 3: $E = \max(E^1, E^2)$
 - 4: $F^1 = F_u - G_e$ ▷ F_u : F of its upper neighbor
 - 5: $F^2 = H_u - G_o$ ▷ H_u : H of its upper neighbor
 - 6: $F = \max(F^1, F^2)$
 - 7: $H = H_{ul} + SM$ ▷ H_{ul} : H of its upper-left neighbor
 - 8: $H = \max(H, E)$
 - 9: $H = \max(H, F)$
 - 10: $H = \max(H, 0)$
 - 11: $A = H$ ▷ A : an auxiliary variable
 - 12: $S = \max(H, S)$ ▷ S : optimal score
-

throughput depends on the CUDA Compute Capability (CC) of each NVIDIA GPU ².

3) *Architectural features on AMD's GPU*: In the RDNA2.0 architecture, the # Cores represent the number of Compute Units (CUs), which are grouped in pairs into Workgroup Processors (WP). On its behalf, AMD calls *wavefront* and *work-item* the equivalent of NVIDIA's *warp* and *thread*, respectively. RDNA2.0 supports both wavefront sizes of 32 and 64 work items but the former is prioritized. Each CU contains two SIMD32 vector units, being able to compute 64 add/subtract/max instructions per cycle (Int32). This means that the instruction throughput is 2 for each work item.

4) *Architectural features on Intel's GPU*: On the discrete segment (dGPUs), Intel has a quite different GPU design philosophy than NVIDIA and AMD. The fundamental block of the Intel Xe microarchitecture is the Xe Core, each of which has 16 Xe Vector Engines (XVEs) ³ that can execute 8 add/subtract/max instructions per cycle (Int32). Thus, Xe Cores and XVEs map to # Cores and # Lanes, respectively, in the proposed model.

On the integrated segment (iGPUs), both Gen9 and Gen12 microarchitectures are similar from a design perspective, differing mainly in the amount of computational resources. In these microarchitectures, the fundamental block is the Subslice, each of which has 8 Execution Units (EUs) that can execute 8 add/subtract/max instructions per cycle (Int32). Thus, Subslices and EUs refer to # Cores and # Lanes, respectively, in the proposed model.

IV. EXPERIMENTAL RESULTS

A. Experimental Design

The experiments were carried out on a set of 10 GPUs, including 6 NVIDIA dGPUs, 1 AMD dGPU, 2 Intel iGPUs, and 1 Intel dGPU. The specific details of these GPUs can be found in Table I. The oneAPI and CUDA versions used were 2022.1.0 and 11.7, respectively. For both CUDA and

²<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#maximize-instruction-throughput>

³Also known as Executions Units (EUs)

TABLE I
GPU SPECIFICATIONS AND THEIR THEORETICAL PEAK PERFORMANCE IN TERMS OF GCUPS

Vendor	NVIDIA						Intel			AMD
Model	GTX 980	GTX 1080	RTX 2070	V100	RTX 3070	RTX 3090	Arc A770	UHD 630	UHD770	RX 6700 XT
Type	Discrete						Discrete	Integrated		Discrete
Microarchitecture	Maxwell (CC 5.2)	Pascal (CC 6.1)	Turing (CC 7.5)	Volta (CC 7.0)	Ampere (CC 8.6)	Ampere (CC 8.6)	Xe HPG	Gen 9.5	Gen 12.2	RDNA 2.0
# Cores	16	20	36	80	46	82	32	3	4	40
# Lanes	32	32	32	32	32	32	16	8	8	32
Instruction throughput	4/2	4/2	2	2	2	2	8	8	8	2
Clock (MHz)	1216	1733	1620	1380	1725	1695	2400	1200	1650	2581
Theoretical peak (GCUPS)	155.648	277.28	311.04	588.8	423.2	741.2	819.2	19.2	35.2	550.61

The instruction throughput for GTX 980 and GTX 1080 is 4 for add/subtract and 2 for max/min. The core instructions include 5 add/subtract and 6 max. Thus the equivalent throughput is 3. The core instruction count for each cell update is 12.

SYCL, the optimization flag `-O3` was used during compilation. To run SYCL code on NVIDIA and AMD GPUs, several modifications had to be made to the build process, as SYCL is not supported by default on these platforms⁴ but Codeplay recently has announced free binary plugins⁵ to support it. After these modifications, it was possible to run DPC++ code on an NVIDIA GPU using the Clang++ compiler (16.0).

SW# was configured with BLOSUM62 as substitution matrix, and 10/2 as insertion/extension gap penalty. The flag `T=0` was also used to remove the impact of the CPU on the final performance (all sequence alignments are computed thoroughly on the GPU).

The performance evaluation was carried out by searching 20 query protein sequences against the well-known Environmental Non-Redundant database (Env. NR) (2021_04 Release), which contains 995210546 amino acid residues in 4789355 sequences, with a maximum length of 16925. Query sequences were selected from the Swiss-Prot database⁶, with lengths ranging from 144 to 5478. The access numbers for these queries are: P02232, P05013, P14942, P07327, P01008, P03435, P42357, P21177, Q38941, P27895, P07756, P04775, P19096, P28167, P0C6B8, P20930, P08519, Q7TMA5, P33450, and Q9UKN1.

In order to minimize fluctuations, the tests were executed 20 times for each set, and the performance was determined based on the average of these multiple runs.

B. Single-GPU Performance and Portability Results

A primary comparison was conducted between the performance of CUDA and SYCL on NVIDIA GPUs (see Fig 2). As can be seen, both programming models achieve practically the same GCUPS values. On the one hand, the largest performance difference in favor of SYCL was observed on the Tesla V100 (3.4%). On the other hand, the CUDA implementation did its part on the GTX 980, outperforming SYCL by 4.6%. Thus,

⁴<https://intel.github.io/llvm-docs/GetStartedGuide.html>

⁵<https://codeplay.com/portal/blogs/2022/12/16/bringing-nvidia-and-amd-support-to-oneapi.html>

⁶Swiss-Prot: <https://www.uniprot.org/downloads>

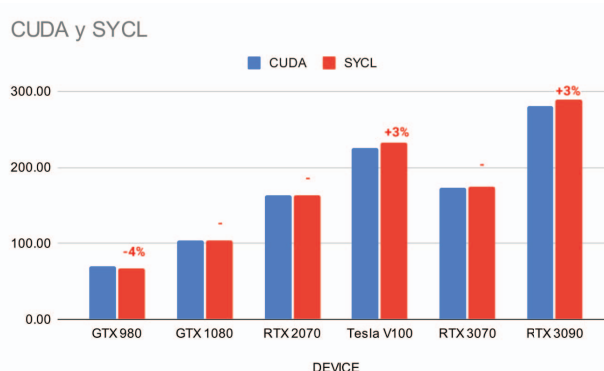


Fig. 2. Performance comparison between CUDA and SYCL on single NVIDIA GPUs

both CUDA and SYCL are capable of delivering comparable performance for this case study on NVIDIA GPUs.

Table II presents a more detailed comparison of the performance and architectural efficiency of CUDA and SYCL codes on NVIDIA, AMD, and Intel GPUs. For each platform, this table shows the peak theoretical performance, the achieved performance for both CUDA and SYCL, and the corresponding architectural efficiency.

On NVIDIA GPUs, CUDA and SYCL demonstrated comparable performance and efficiency values, as was already noted in the analysis from Fig. 2. As expected, more powerful GPUs are able to achieve higher GCUPS values. As for the architectural efficiency values, they are in the range of 37%-52%. It is important to note that, although the highest GCUPS value is presented by RTX 3090 GPU, the most efficient one turns out to be RTX 2070 GPU.

For AMD and Intel GPUs, only the results for SYCL are shown, due to CUDA just supports NVIDIA GPUs. This fact highlights the already mentioned greater portability of SYCL over CUDA. It can be said that the results of the SYCL version on these GPUs are generally good. On the one hand, SYCL matches its best efficiency rate on NVIDIA GPUs when running on AMD GPUs. On the other hand, SYCL beats

TABLE II
GCUPS AND ARCHITECTURAL EFFICIENCIES OF BOTH CUDA AND SYCL CODES ON SINGLE GPUS.

Vendor	Platform		CUDA		SYCL	
	GPU	GCUPS peak	GCUPS ach.	Arch eff.	GCUPS ach.	Arch eff.
NVIDIA	GTX 980	155.5	70.6	45.3%	67.7	43.5%
	GTX 1080	277.2	104.5	37.7%	103.8	37.4%
	RTX 2070	311.0	162.5	52.2%	163.1	52.4%
	Tesla V100	588.8	224.9	38.2%	233.0	39.5%
	RTX 3070	423.2	173.1	40.9%	174.4	41.2%
	RTX 3090	741.3	280.2	37.8%	288.6	38.9%
Intel	Arc A770	819.2	×	NA	191.4	23.3%
	UHD 630	19.2	×	NA	13.1	68.4%
	UHD 770	35.2	×	NA	26.6	75.7%
AMD	RX 6700 XT	550.6	×	NA	284.4	51.7%

that mark on the 2 integrated GPUs, achieving up to +23.1% architectural efficiency. The only negative aspect is SYCL’s performance on Intel’s Arc A770, where performance drops to 23.3% of architectural efficiency. This value represents its lowest performance and the cause could be related to Intel’s discrete GPU design philosophy, which differs from NVIDIA and AMD. However, we plan to profile the code to learn more about this issue.

TABLE III
PERFORMANCE PORTABILITY OF BOTH CUDA AND SYCL CODES ON SINGLE GPUS.

Platform set (H)	$\Phi(\alpha, p, H)$	
	CUDA	SYCL
NVIDIA	42%	42.2%
AMD	NA	51.7%
Intel (discrete)	NA	23.3%
Intel (integrated)	NA	72.0%
Intel (all)	NA	55.8%
$\overline{\text{NVIDIA}} \cup \overline{\text{AMD}}$	$\overline{\text{NA}}$	44.3%
$\text{NVIDIA} \cup \text{Intel}$	NA	47.2%
$\text{Intel} \cup \text{AMD}$	NA	54.8%
$\overline{\text{NVIDIA}} \cup \overline{\text{AMD}} \cup \overline{\text{Intel}}$	$\overline{\text{NA}}$	47.2%

The performance portability of both CUDA and SYCL codes is evaluated in Table III, where it can be noted that aggregated results are consistent with those observed on an individual basis before. For NVIDIA GPUs, the performance portability of both is quite similar, with values of 42% and 42.2%, respectively. As seen before, this indicates that both programming models can deliver a consistent level of performance across the different NVIDIA GPUs used in the tests.

In the case of Intel GPUs, SYCL demonstrated very good architectural efficiency values on the iGPUs, in contrast to the lower efficiency exhibited on the dGPU. Moreover, when considering the combination of AMD and Intel GPUs, SYCL achieves the highest performance portability of the middle set.

However, the performance portability decreases when NVIDIA GPUs are also included (last set), as SYCL performance is lower on these devices.

Building on the previous analysis, SYCL consistently outperforms CUDA in terms of performance portability in this study. To be more precise, SYCL achieved nearly the same architectural efficiency as CUDA considering 6 NVIDIA GPUs with 5 different microarchitectures. Moreover, SYCL was not only able to run on multiple vendor GPUs (AMD and Intel), but its architectural efficiency was superior in 3 of the 4 cases tested. This demonstrates not only SYCL’s broad compatibility but also its capability to improve performance across a diverse range of GPUs for this application.

C. Multi-GPU Performance and Portability Results

To complement the previous single-GPU analysis, a performance comparison was carried out between CUDA and SYCL using different multiple NVIDIA GPUs (see Fig. 3). As is the single-GPU case, the two programming models achieve practically the same GCUPS values when NVIDIA devices are used, for both homogeneous and heterogeneous multi-GPU configurations. While CUDA outperforms SYCL when using $2 \times \text{GTX1080}$ by approximately 1%, SYCL achieves the best performance in all other cases, achieving up to 5% higher GCUPS. Therefore, it can be noted that SYCL does not imply additional overhead when multiple GPUs are used.

Table IV presents a more detailed comparison of the performance and architectural efficiency of CUDA and SYCL codes on 5 different multi-GPU configurations. It can be seen that for NVIDIA multi-GPUs, the efficiency rates achieved when using 2 GPUs combined are a bit lower than when using a single GPU. This behavior occurs in 3 of the 4 configurations tested (the exception is when using $2 \times \text{Tesla V100}$) and can be explained by 2 reasons. On the one hand, it is usual that the efficiency decreases when fixing the problem size and increasing the amount of computational resources. On the other hand, the workload distribution strategy of $SW\#$ is very simple, since it distributes the query sequences among the GPUs and does not consider each GPU computing power. Because these sequences do not have the same length, load imbalance can occur between GPUs, reducing performance.

Finally, SYCL once again demonstrates its increased functional portability with Intel’s multi-GPU case. While the performance is not good for the aforementioned reasons, it is interesting to note how SYCL allows using 2 Intel GPUs of different types at the same time: an iGPU and a dGPU.

V. RELATED WORKS

Some preliminary studies have compared the performance between SYCL and CUDA in different domains. In [24], the authors employed ADEPT, a GPU-accelerated short-read alignment kernel, as a case study. They found that the SYCL implementation runs approximately $2 \times$ slower than its CUDA counterpart in all experiments when using an NVIDIA V100 GPU. The authors attribute this discrepancy to CUDA’s superior utilization of memory cache and SYCL’s greater reliance

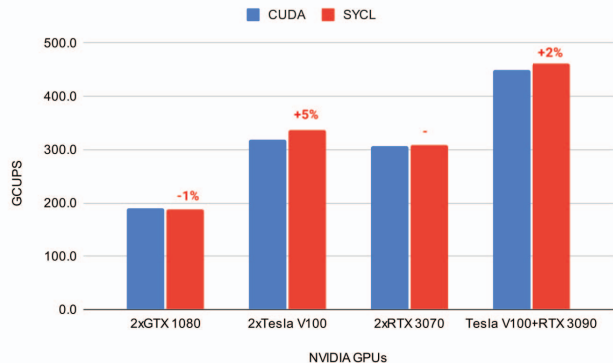


Fig. 3. Performance comparison between CUDA and SYCL on multiple NVIDIA GPUs

TABLE IV
GCUPS AND ARCHITECTURAL EFFICIENCIES OF BOTH CUDA AND SYCL CODES ON MULTIPLE GPUS.

Vendor	Platform		CUDA		SYCL	
	GPUs	GCUPS peak	GCUPS ach.	Arch eff.	GCUPS ach.	Arch eff.
NVIDIA	2× GTX 1080	554.6	189.8	34.2%	187.8	33.9%
	2× Tesla V100	846.4	318.1	27.0%	336.5	28.6%
	2× RTX 3070	1177.6	306.5	36.2%	308.9	36.5%
	Tesla V100 + RTX 3090	1330.1	450.5	33.8%	460.7	34.6%
Intel	Arc A770 + UHD 770	854.4	×	NA	126.8	14.8%

on register usage. Additionally, the authors verified SYCL’s code portability on an Intel P630 GPU.

In [25], the authors delve into the process of migrating a CPU+GPU application for epistasis detection from CUDA to SYCL, founding that the highest performance of both versions is comparable on an NVIDIA V100 GPU. However, it is important to remark that some hand-tuning was required in the SYCL implementation to reach its maximum performance. When investigating the PTX code, the authors noted that SYCL does not perform the same optimizations as CUDA, such as loop unrolling.

In [26], the authors identified performance gaps in several bioinformatics applications. The study involved the selection of open-source applications that had been migrated from CUDA to SYCL, followed by a comprehensive evaluation of their performance on an NVIDIA V100 GPU. Through profiling analysis, the authors found that the SYCL compiler lacks certain optimizations that the CUDA version does, including memory management, instruction vectorization, and loop unrolling, among others.

In [27], a performance comparison is carried out between SYCL and CUDA in the context of AI models. The authors extend the SYCL-DNN library to include support for NVIDIA GPUs using DPC++ and evaluate its performance against the optimized cuDNN library. Initially, they observed that the non-optimized SYCL-DNN is approximately 50% slower than cuDNN due to a poorly optimized implementation of SYCL for local memory. However, after using SYCL-BLAS, a significant speedup of up to 90% of cuDNN’s performance is achieved. The remaining 10% difference is attributed to hand-written, optimized implementations in CUDA.

In [28], the authors compare two CUDA and SYCL versions of the AutoDock-GPU molecular docking application on an Intel Xeon Platinum 8360Y CPU, an NVIDIA A100 GPU, and an Intel Max 1550 GPU. On the A100 GPU, SYCL exhibits slower performance compared to CUDA in some cases, with performance ratios ranging from $1.24\times$ to $2.38\times$. However, in the small test cases, SYCL outperforms CUDA by $1.09\times$. The authors attribute the lower ratios to the synchronization effort required in compute-intensive regions like the scoring function and gradient calculation. They highlight the need for deeper performance analysis and suggest further optimization, particularly in compute-intensive areas, to improve SYCL performance.

In [29], the authors analyze the performance of mini-apps that have been created in both SYCL and CUDA, running on an NVIDIA V100 GPU. Even though there are some features not fully supported, SYCL performance is comparable to that of CUDA. Moreover, the performance differences largely stem from variations in memory access patterns.

In [30], the author evaluate the gap between performance and code portability in HPC accelerators using the well-known k-means algorithm comparing SYCL with CUDA and OpenMP. SYCL implementation reports higher performance on Intel GPUs and CPUs, equivalent performance on NVIDIA GPUs, and offers potential multi-vendor compatibility.

Unlike the previous works and beyond the results obtained, this performance portability study has considered different GPU architectures, including single and multi-GPU configurations from multiple vendors. To the best of our knowledge, no study has considered such a diverse and large set of GPUs.

VI. CONCLUSIONS AND FUTURE WORK

In the field of heterogeneous computing, ensuring functional portability is not trivial for a programming language, and thus providing performance portability represents an even greater challenge. In this study, we address this issue by assessing the portability and performance of the SYCL and CUDA languages for the Smith-Waterman protein database search across different GPU architectures from multiple vendors. The experimental results show that CUDA and SYCL are capable of delivering comparable performance for this case study on NVIDIA GPUs, including single and multi-GPU configurations. When moving to AMD and Intel GPUs, SYCL was not only able to run on these devices, but its architectural efficiency was superior in 3 of the 4 cases tested. This

demonstrates not only SYCL's broad compatibility but also its capability to improve performance across a diverse range of GPUs for this application.

Since SYCL is still an immature programming model, the positive results found here cannot be generalized; performance will largely depend on the characteristics of the application and the capabilities of the compilers. However, they are a sample of the promising opportunities that SYCL can offer for heterogeneous computing.

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 alignment [22], such as instruction reordering to reduce their count and the use of lower precision integers to increase parallelism⁷. Additionally, improving the workload distribution strategy when using more than one GPU. These improvements will lead to higher efficiency rates.
- Running the SYCL code on other architectures (such as CPUs and CPUs+GPUs) and also considering other SYCL implementations (such as OpenSYCL and ComputeCPP), as well as other programming models like Kokkos⁸ and RAJA⁹, to strengthen the current performance portability study.

REFERENCES

- [1] H. Giefers, P. Staar, C. Bekas, and C. Hagleitner, "Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of gpu, xeon phi and fpga," in *2016 IEEE ISPASS*, 2016, pp. 46–56.
- [2] A. Marowka, "Reformulation of the performance portability metric," *Software: Practice and Experience*, vol. 52, no. 1, pp. 154–171, 2022.
- [3] M. S. Nobile, P. Cazzaniga, A. Tangherloni, and D. Besozzi, "Graphics processing units in bioinformatics, computational biology and systems biology," *Briefings in Bioinformatics*, vol. 18, no. 5, pp. 870–885, 07 2016.
- [4] M. Korpar and M. Sikic, "SW# - GPU-enabled exact alignments on genome scale," *Bioinformatics*, vol. 29, no. 19, pp. 2494–2495, 2013.
- [5] M. Korpar, M. Susic, D. Blazeka, and M. Sikic, "SWdb: GPU-Accelerated Exact Sequence Similarity Database Search," *PLOS ONE*, vol. 10, no. 12, pp. 1–11, 12 2016.
- [6] M. Costanzo, E. Rucci, C. García-Sánchez, M. Naiouf, and M. Prieto-Matías, "Migrating cuda to oneapi: A smith-waterman case study," in *Bioinformatics and Biomedical Engineering*, I. Rojas, O. Valenzuela, F. Rojas, L. J. Herrera, and F. Ortuño, Eds. Cham: Springer International Publishing, 2022, pp. 103–116.
- [7] H. Lan, W. Liu, Y. Liu, and B. Schmidt, "SWhybrid: A hybrid-parallel framework for large-scale protein sequence database search," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 2017, pp. 42–51.
- [8] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.
- [9] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science and Engg.*, vol. 12, no. 3, p. 66–73, may 2010.
- [10] "The OpenMP Specification," <http://https://www.openmp.org/>.
- [11] "The OpenACC Specification," <http://https://www.openacc.org/>.
- [12] R. Farber, *Parallel Programming with OpenACC*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.
- [13] Khronos SYCL working group, "Sycl 2020 specification," <https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html>, 2023.
- [14] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openacc — first experiences with real-world applications," in *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 859–870.
- [15] Codeplay Software, "Computecpp community edition," <https://developer.codeplay.com/products/compute/cpp/ce/home>, 2023.
- [16] B. Ashbaugh, A. Bader, J. Brodman, J. Hammond, M. Kinsner, J. Pennycook, R. Schulz, and J. Sewall, "Data parallel c++ enhancing sycl through extensions for productivity and performance," in *Proceedings of the International Workshop on OpenCL*, 2020, pp. 1–2.
- [17] "The triSYCL project," <https://github.com/triSYCL/triSYCL>, 2023.
- [18] Aksel Alpay, "Opensycl implementation," <https://github.com/OpenSYCL/OpenSYCL>, 2023.
- [19] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, March 1981.
- [20] O. Gotoh, "An improved algorithm for matching biological sequences," in *Journal of Molecular Biology*, vol. 162, 1981, pp. 705–708.
- [21] E. F. De Oliveira Sandes, A. Boukerche, and A. C. M. A. De Melo, "Parallel optimal pairwise biological sequence comparison: Algorithms, platforms, and classification," *ACM Comput. Surv.*, vol. 48, no. 4, 2016.
- [22] T. Rognes, "Faster Smith-Waterman database searches with inter-sequence SIMD parallelization," *BMC Bioinformatics*, vol. 12:221, 2011.
- [23] S. Pennycook, J. Sewall, and V. Lee, "Implications of a metric for performance portability," *Future Generation Computer Systems*, vol. 92, pp. 947–958, 2019.
- [24] M. Haseeb, N. Ding, J. Deslippe, and M. Awan, "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)*, 2021, pp. 68–78.
- [25] Z. Jin and J. S. Vetter, "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*, ser. BCB '22. New York, NY, USA: ACM, 2022.
- [26] —, "Understanding performance portability of bioinformatics applications in sycl on an nvidia gpu," in *2022 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 2022, pp. 2190–2195.
- [27] M. Tanvir, K. Narasimhan, M. Goli, O. El Farouki, S. Georgiev, and I. Ault, "Towards performance portability of ai models using sycl-dnn," in *International Workshop on OpenCL*, ser. IWOCCL'22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: <https://doi.org/10.1145/3529538.3529999>
- [28] L. Solis-Vasquez, E. Mascarenhas, and A. Koch, "Experiences migrating cuda to sycl: A molecular docking case study," in *Proceedings of the 2023 International Workshop on OpenCL*, ser. IWOCCL '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: <https://doi.org/10.1145/3585341.3585372>
- [29] B. Homerding and J. Tramm, "Evaluating the performance of the hipsycl toolchain for hpc kernels on nvidia v100 gpus," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCCL '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: <https://doi.org/10.1145/3388333.3388660>
- [30] None, "Exploring the performance and portability of the k-means algorithm on SYCL across CPU and GPU architectures," *The Journal of Supercomputing*, pp. 1–27, 2023. [Online]. Available: <https://doi.org/10.1007/s11227-023-05373-2>