The Journal of Supercomputing (2024) 80:14203–14223
https://doi.org/10.1007/s11227-024-05957-6

SYCL in the edge: performance and energy evaluation for heterogeneous acceleration

Youssef Faqir-Rhazoui[1] · **Carlos García**[1]

Accepted: 3 February 2024 / Published online: 16 March 2024

Abstract
Edge computing is essential to handle increasing data volumes and processing capacities. It provides real-time and secure data processing near data sources, like smart devices, alleviating cloud computing energy use, and saving network bandwidth. Specialized accelerators, like GPUs and FPGAs, are vital for low-latency edge computing but the requirements to customized code for different hardware and vendors suppose important compatibility issues. This paper evaluates the potential of SYCL in addressing code portability issues encountered in edge computing. We employed the Polybench suite to compare various SYCL implementations, specifically DPC++ and AdaptiveCpp, with the native solution, CUDA. The disparity between SYCL implementations was negligible, at just 5%. Furthermore, we evaluated SYCL in the context of specific edge computing applications such as video processing using three different optical flow algorithms. The results revealed a slight performance gap of 3% when transitioning from CUDA to SYCL. Upon evaluating energy consumption, the observed difference ranged from ±10%, depending on the application utilized. These gaps are the price one may need to pay when achieving the ability to successfully run the same code on two distinct edge boards. These findings underscore SYCL's capacity to increase productivity in terms of development costs and facilitate IoT deployment without being locked into a particular platform or manufacturer.

Keywords SYCL · CUDA · Edge computing · Polybench · Jetson · Optic flow

✉ Youssef Faqir-Rhazoui
yelfaqir@ucm.es

Carlos García
garsanca@ucm.es

[1] Department of Computer Architecture and Automatics, Complutense University of Madrid, Madrid, Spain

1 Introduction

Edge computing has emerged as a crucial technology due to the growing volume of data and processing demands. Edge computing operates in close proximity to data sources [1, 2], like smart devices, storing and processing data at the network's edge. It offers fast, real-time, and secure data processing [3], addressing issues like energy consumption in cloud computing, cost reduction, and network bandwidth relief. The increasing prominence of IoT [4] has transformed edge computing into a highly discussed subject, presenting ongoing challenges [5–7] such as selecting the most suitable platform to achieve among others real-time data processing near the data source and ensuring robust data privacy. Nevertheless, one of the foremost challenges persisting in the deployment of IoT systems is the imperative of achieving reduced energy consumption [8] while concurrently upholding robust computational capabilities essential for supporting real-time AI or ML applications.

To address these handicaps, there is a growing trend toward the adoption of accelerators, collectively referred to as xPU (including GPUs, FPGAs, SoCs, and more), which substantially reduce power footprint [9] when compared to general-purpose CPUs. However, employing accelerator languages designed for specific hardware architectures introduces compatibility obstacles meanwhile a custom code for each device (e.g., CUDA, VHDL, etc.) is imperative. The industry's motivation to progress in this direction is compounded by two significant challenges: firstly, to select the most suitable system from a huge plethora of devices with notable architectural differences, and secondly, the absence of a universally accepted programming standard. Under this premise, we can highlight recent advances with the creation of the Unified Acceleration (UXL) Foundation,[1] announced by the Linux Foundation on September 2023, which proposes oneAPI [10] and SYCL [11] programming as an open-source specification to support a common code base capable of running across multiple architectures.

Until now, native accelerator languages have empowered programmers to deploy code tailored for specialized hardware devices like GPUs, FPGAs, or ASICs. These languages mostly proprietary APIs are engineered to enhance the performance and efficiency of compute-intensive applications. Nevertheless, a common challenge faced by most accelerator languages is their propensity to disrupt compatibility among different hardware architectures. For instance, CUDA [12] is tailored for NVIDIA GPUs, HIP [13] for AMD GPUs, or VHDL for FPGAs.

In contrast, SYCL [11] is a versatile programming model and standard that empowers developers to create heterogeneous parallel code based on ISO C++. SYCL streamlines the process by allowing programmers to write code once, which can then seamlessly execute across multiple vendor CPUs, GPUs, and FPGAs via OpenCL. What sets SYCL apart is its compatibility with modern C++ features like templates, lambdas, and exceptions, which facilitate the expression of parallelism and data movement. SYCL's remarkable versatility not only facilitates the development of portable applications for diverse heterogeneous edge computing systems

[1] Unified Acceleration (UXL) Foundation: https://uxlfoundation.org/

[14], including CPUs, GPUs, and FPGAs, but also serves as a foundational tool for implementing cost-effective exploration methodologies aimed at reducing development complexity. By employing a unified development approach across multiple edge computing platforms, it becomes possible to discern the architecture that best suits specific problem domains, especially those reliant on critical factors such as power efficiency, cost-effectiveness, and real-time performance requirements.

Moreover, SYCL has been extensively tested on HPC environments and compared with other programming languages such as CUDA, OpenMP, or OpenCL [15–17]. While the utilization of SYCL in the realm of edge computing remains relatively unexplored [18] apart from preliminary experiments of porting CUDA codes [19]. We believe that its adoption holds significant potential for achieving performance portability. In this paper, we assess the effectiveness of SYCL on two edge computing boards. We employ a suite of benchmarks to verify SYCL's compatibility across different architectures with a special interest in performance and energy consumption. Furthermore, we explore the portability of various motion estimation-based vision algorithms, incorporating accelerators from different vendors.

The following paper is organized as follows. Section 2 introduces the SYCL language and program architecture. In Sect. 3, the benchmarks used in this study are discussed. Section 4 focuses on the environment configuration and experiment methodology used. In Sect. 5, the experiments and results achieved are presented. In section 6, an experiment discussion is performed. And finally, the Sect. 7 concludes with the main remarks.

2 The SYCL paradigm in a nutshell

SYCL is a standard (SYCL 2020) developed and maintained by the Khronos Group, similar to other standards such as OpenMP (e.g., 4.5, 5.1, etc.) or OpenCL (e.g., 2.1, 3.0, etc.) [20–22]. Its main purpose is to enable developers to use any ISO C++ compiler (e.g., GCC, Clang, NVCC, ICC, etc.), and utilize C++ lambdas to encapsulate device kernel execution. SYCL does not aim to replace other parallel models or backends (e.g., CUDA, HIP, OpenCL, etc.) but rather to complement them. Since all these models are C++-compatible, SYCL uses C++ lambdas to extend the native API of different backends. For instance, when allocating memory on an NVIDIA GPU, a SYCL memory allocation automatically triggers a native CUDA allocation at background. Then, you can consider SYCL as the facade design pattern, which serves as a front-facing interface to other backends [23].

Up to this point, we have solely addressed the SYCL standard; however, it is crucial to recognize that SYCL does not have a singular implementation. The most feature-rich implementation is Intel Data Parallel C++ (DPC++) [14], which not only conforms to the SYCL 2020 standard but also includes other custom features.[2] The Intel oneAPI DPC++/C++ compiler known as DPC++ is a compiler-based implementation, based on the Clang/LLVM project. It is important to remark that

[2] These features are typically incorporated into new SYCL releases. However, we did not use them in this paper.

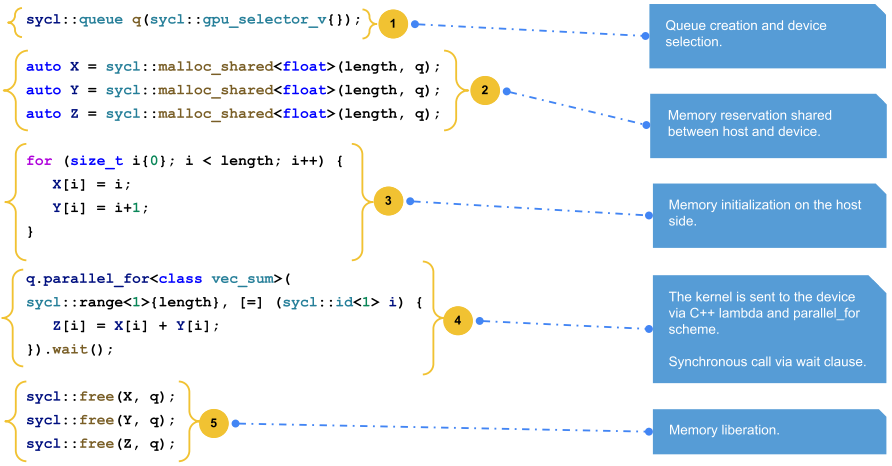

Fig. 1 SYCL piece of code performing a vector addition

DPC++ compiler is open source, although Intel also offers a commercial alternative available on the oneAPI toolkits. The oneAPI includes additional tools such as profiler and optimized libraries. While custom features are present in DPC++, we opted not to employ them in our study, with the aim of ensuring the portability of our developments.

The other noteworthy implementation is AdaptiveCpp previously known as hipSYCL, which is a library-based implementation. This means that they have developed a C++ library and rely on third-party compilers. It is generally recommended to use it with the stock Clang/LLVM compiler, which was designed to support CUDA, HIP, OpenMP, and OpenCL source codes [24, 25].

The Clang/LLVM compiler is responsible for compiling SYCL code, and it performs different steps such as front-end, middle-end, and back-end. In the front-end phase, the compiler separates the host code from the device code, while in the middle-end phase, it transforms the device code into an intermediate representation known as LLVM IR. The back-end stage then compiles the LLVM IR representation into the device's native code and combines everything into a final file called "fat binary." This compiler can generate a final binary that can run on multiple devices, including multi-vendor GPUs or even FPGAs, as described in [10, 26].

Figure 1 illustrates a basic SYCL program that sums two vectors into a third one. The usual SYCL scheme begins by creating a queue associated with the target device (step 1). The queue receives the following kernels and is responsible for placing them on the device based on a policy. Additionally, we can allocate the program memory (step 2), which in this particular example is shared between the host and the device. This feature allows the host to be able to initialize the memory data on its side without any other restrictions (step 3), and later being used by the device without any explicit data movement.

The next step is to invoke the kernel execution on the device (step 4). SYCL supports multiple parallel patterns, but in this code example, the *parallel_for* scheme

is performed, specifying the problem size (length) so the kernel launches length instances or threads. In SYCL by default, the kernel launch is asynchronous so it is mandatory to add the *wait()* clause to maintain the execution coherence. Finally, the memory is freed with the corresponding call because it is tied to the queue (step 5).

3 Benchmarking SYCL in edge platforms

SYCL was tested on both conventional and HPC systems, as the next subsection highlights. However, there is a dearth of literature regarding the potential use of SYCL on the edge. We considered the use of benchmark suites developed for or adapted to SYCL, as they would favor direct comparisons with other programming models like CUDA, and also permit the compilation with various SYCL implementations, including DPC++ and AdaptiveCpp.

3.1 SYCL benchmark suits

When it comes to benchmarks, there are several suites available for SYCL. The Rodinia[3] benchmarks are implemented in multiple languages, including SYCL, and encompass a wide range of field benchmarks, such as medical imaging or image compression [15, 27].

XSBench[4] is a benchmark suite designed to evaluate the performance of Monte Carlo neutron transport codes used in the field of nuclear engineering and reactor physics. The benchmark suite provides a set of representative problems that simulate the behavior of neutrons in a nuclear reactor. These problems cover a range of materials, geometries, and physics phenomena to assess the performance of different Monte Carlo codes accurately [28].

On its side, HeCBench[5] is a large collection of heterogeneous programming models such as (SYCL, OpenCL, CUDA, etc.). HeCBench recollects benchmarks from many sources, including many of Rodinia or XSBench [29]. The suit includes benchmarks in the area of lineal algebra, AI, or machine learning.

Polybench[6] consists of a set of computationally intensive kernels that represent common algorithmic patterns found in scientific and engineering applications, such as linear algebra computations, image processing, stencil computations, and more. These kernels are implemented in C, CUDA, and OpenMP among other programming languages, and are designed to be representative of real-world workloads [30].

SYCL-Bench[7] provides a set of benchmark kernels and applications that cover a range of common parallel computing patterns and algorithms. These benchmarks are implemented using SYCL and are designed to evaluate the performance of

[3] Rodinia code migrated to SYCL through oneAPI: https://github.com/artecs-group/rodinia-dpct-dpcpp

[4] https://github.com/ANL-CESAR/XSBench

[5] https://github.com/zjin-lcf/HeCBench/tree/master

[6] https://github.com/sgrauerg/polybenchGpu

[7] https://github.com/unisa-hpc/sycl-bench

SYCL compilers, runtime systems, and underlying hardware architectures. SYCL-Bench also integrates fifteen kernels/applications from Polybench. This suite also has the possibility to execute on different SYCL implementations, such as DPC++, ComputeCpp, triSYCL, and AdaptiveCpp [31].

3.2 Image processing for optic flow

Optical flow, a crucial component in machine vision systems, calculates a dense field of displacement vector which represents the pixel motion [32] of adjacent frames in consecutive image frames. It holds a pivotal significance in applications of image processing such as video coding, tracking, autonomous driving, or biomedical imaging. It is based on finding the apparent motion of objects in a sequence of images from a camera, extracting a two-dimensional vector related to the object's motion.

In recent decades, significant advancements in optical flow estimation have been fueled by two main factors. First, the emergence of advanced-level datasets [33–35] has led to continuous improvements in optical flow algorithms. Second, the growing computational resources available in modern microchips such as GPUs accelerators have pushed the development of novel strategies rooted in deep learning approaches.

Horn and Schunck (HS)[36] pioneered the initial optical flow estimation proposal, employing a variational method that leveraged both brightness constancy and spatial smoothness assumptions. It is based on applying spatial and temporal derivatives [37] to the intensity of the image to extract the optical flow vector by solving a multi-dimensional system of equations. To speedup the convergence, hierarchy processing techniques can also be applied [37, 38]. An implementation of the CUDA Horn–Schunck method can be found in the CUDA toolkit examples,[8] and it has recently been ported to SYCL using an automatic compatibility tool available on the Intel's oneAPI suite.[9]

Subsequently, the Lucas and Kanade (LK) method [39], proposed by Bruce D. Lucas and Takeo Kanade, is based on the premise that optical flow remains largely consistent within the immediate vicinity of the analyzed pixel. This technique involves solving the core optical flow equations for all pixels within this local neighborhood through the application of the least squares criterion.

While HS and LK represent the current state of the art in optical flow techniques and have been used as benchmarks to evaluate ad hoc implementations in several platforms based on GPUs, FPGAs, or DSPs [40–42], they still are pertinent in the embedded system scope. However, it is worth noting that numerous research endeavors have since addressed issues such as high-speed object detection, occlusion handling, illumination changes, and noise reduction. This underscores the community's commitment to enhancing these techniques [43]. A notable proposal that has garnered significant attention from researchers is the TV-L[1] method by Zach et al. [44–46], which employs a variational approach to tackle challenges such as

[8] HSOpticalFlow - Optical Flow: https://github.com/NVIDIA/cuda-samples/tree/master/Samples/5_Domain_Specific/HSOpticalFlow

[9] Migrating the HSOpticalFlow Estimation from CUDA to SYCL: https://www.intel.com/content/www/us/en/developer/articles/technical/migrating-hsopticalflow-from-cuda-to-sycl.html

Table 1 Technical specifications of the Jetson Orin Nano and Up Squared Pro 7000 Edge

		NVIDIA Jetson Orin Nano	UP Squared Pro 7000 Edge
CPU	Name	ARM Cortex-A78AE	Intel Atom X7425E
	Frequency	up to 1.5 GHz	1.50 GHz (Base) 3.40 GHz (Boost)
	Cores	6	4
GPU	Name	NVIDIA Ampere GPU	Intel UHD Graphics Gen 12
	Frequency	625MHz	1 GHz
	Cores	1,024 CUDA cores	24 execution units
	Perf. (FP32)	1,280 GFLOPS	460.8 GFLOPS
	Driver	JetPack 5.1.2	23.35 (OpenCL)
Memory		8 GB	8 GB
Storage		SD Card Slot & external NVMe via M.2 Key M (not included)	64 GB eMMC
Power Consumption		7W / 15W	12W
Price		499$	399$

illumination changes, outliers, and flow discontinuities. Other studies, such as those cited in references [47, 48], provide evidence of its advantageous trade-offs on embedded hardware.

4 Methods

This section briefly describes the configuration and methodology used for the experimentation.

4.1 Environment configuration

Table 1 summarizes the main characteristics of the boards used in this research: the Nvidia Jetson Orin Nano[10] and the UP Squared Pro 7000 Edge.[11] While the first system is based on a SoC equipped with an ARM CPU (Cortex-A78AE) and an NVIDIA Ampere GPU, the second one is based on a SoC equipped with an Intel Atom X7425E and a UHD Graphics Gen 12 GPU.

Despite the Nvidia Jetson Orin Nano can be configured to operate at either seven or fifteen watts, it has been set to fifteen watts. In contrast, the power consumption is not configurable at the UP Squared Pro 7000 Edge board, and it works at twelve watts. To measure power consumption, we used *tegrastats* in the Jetson board, while *turbostats* were employed in the other device.

[10] Orin Nano specs:https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/
[11] UP Pro 7000 specs:https://www.mouser.es/new/aaeon-up/aaeon-up-pro-7000-boards

Regarding the software configuration, we utilized two SYCL flavors: DPC++ and AdaptiveCpp. DPC++ can be built from scratch following the instructions from the Intel public repository.[12] As oneAPI is primarily designed for x86/64 architecture, we underwent the process of compiling the DPC++ compiler for ARM architecture, which was then applied to the Orin Nano board.[13] From using AdaptiveCpp,[14] it is necessary to build from the sources on both boards.

Two more aspects worth mentioning regarding SYCL implementations. Meanwhile, SYCL implementations prioritize the portability of the developed codes for running on various devices, SYCL implementation accomplishes this task in different manners. For instance, DPC++ utilizes OpenCL to run on multicore CPUs, while AdaptiveCpp exploits parallel facilities by means of OpenMP. It is noteworthy to remark that although OpenMP enhances compatibility with non-x86 architectures, it may lead to reduced performance compared to OpenCL [49, 50]. In contrast, DPC++ restricts execution on ARM-based CPUs due to the lack of official OpenCL support. Lastly, regardless of OpenCL or Intel Level0 backends in the current state of AdaptiveCpp makes impossible its support on Intel GPUs.

4.2 Benchmarking methodology

To assess the performance portability of SYCL, we evaluate both CPU and GPU performance, as the same code can run on both devices. Additionally, we compare the performance of SYCL against the native CUDA code in the Jetson Orin GPU.

Since SYCL-Bench has a specific SYCL benchmarks suite, we have just selected a subset known as the Polybench benchmarks to perform the comparison between SYCL and CUDA. In particular, we choose the Polybench suite for CUDA evaluation.

In order to make as fair a comparison as possible, we kept all the default parameter configurations for the tests. Table 2 provides an overview of the benchmark descriptions and the parameters established for the benchmarking.

The assessment of energy consumption utilized common AI algorithms, also employed in edge computing, was acquired from the HeCBench suite. Table 3 provides a description of these algorithms.

With the purpose of evaluating modern embedded systems in a more realistic scenario, we choose a workload associated with computer vision as a case study. This experimentation is based on the evaluation of the performance of relevant motion estimation algorithms such as LK, HS, and TV-L^1.

The LK algorithm was developed from scratch. Although HS CUDA and SYCL implementations were inspired from the previously mentioned sources, it was necessary to update them to comply with the SYCL 2020 standard. In the case of the TV-L^1 algorithm, no sources were found except for the OpenMP implementation[15]

[12] DPC++ compiler: https://github.com/intel/llvm/blob/sycl/sycl/doc/GetStartedGuide.md
[13] Both boards use oneAPI 2023.2.
[14] https://github.com/AdaptiveCpp/AdaptiveCpp/blob/develop/doc/installing.md
[15] https://www.ipol.im/pub/art/2013/26/

Table 2 Polybench suite description and the input parameter size used

Area	Benchmark	Size	Description
Convolution	2DConv	4096	2D convolution
	3DConv	512	3D convolution
Linear Algebra	2 mm	1024	2 Matrix Multiplications (D=A.B; E=C.D)
	3 mm	512	3 Matrix Multiplications (E=A.B; F=C.D; G=E.F)
	Atax	4096	Matrix Transpose and Vector Multiplication
	Bicg	16384	BiCG Sub Kernel of BiCGStab Linear Solver
	Gemm	1024	Matrix-multiply C=alpha.A.B+beta.C
	Gesummv	16384	Scalar, Vector, and Matrix Multiplication
	Gramschmidt	1024	Gram–Schmidt decomposition
	Mvt	16384	Matrix Vector Product and Transpose
	Syr2k	1024	Symmetric rank-2k operations
	Syrk	1024	Symmetric rank-k operations
Datamining	Correlation	1024	Correlation Computation
	Covariance	1024	Covariance Computation
Stencil	Fdtd2d	1024	2D Finite Different Time Domain Kernel

Table 3 HeCBench AI benchmarks description and parameter specification

Benchmark	Parameters	Description
Attention multi-head	P1: 100000	Mechanism to focus on various aspects of input data to capture diverse relationships and dependencies
Dense embedding	P1: 10000 P2: 64 P3: 1000	Refers to the representation of objects, or words in a continuous vector space, commonly employed in natural language processing for capturing semantic relationships
ReLU	P1: 10000000 P2: 800	Activation function used in neural networks
ResNet	P1: 5 P2: 300	Residual network is a type of deep neural network architecture

from this work [46]. Both the LK and TV-L[1] algorithms were ported to SYCL using the SYCLomatic tool and later fine-tuned to enhance performance and readability.

To assess this study, we selected a recognized suite of datasets widely used in the field of optical flow. The key characteristics of the datasets used are outlined below:

- Schoolgirls with an image resolution of 432×240 pixels found at.[16]
- Middlebury dataset [33] includes a twelve scenes with images of 640×480.
- MPI-Sintel [34] is a synthetic dataset based on an animation film which contains frames of 1024×436 size.

[16] https://github.com/hitachinsk/FGT

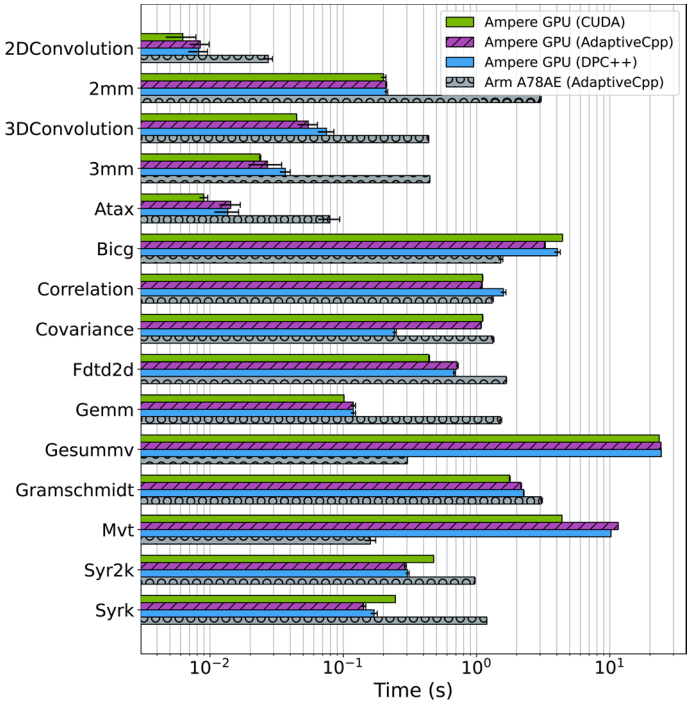

Fig. 2 Execution time recorded for Polybench suite tests on the Jetson Orin Nano using CUDA and SYCL

5 Experimental results

This section presents the results achieved from the Polybench suite, optical flow methods, and HeCBench AI subset. We have divided this section into three parts, each dedicated to one of the experiments.

5.1 Polybench experiments

Figure 2 illustrates the execution times obtained from Polybench on the Jetson Orin Nano board. It includes the execution on GPU devices using the CUDA programming model, AdaptiveCpp, and DPC++ for SYCL as well as on the ARM A78AE CPU through AdaptiveCpp. It is worth noting that the DPC++ cannot be used on the CPU due to the absence support of for OpenCL on ARM processors. In more detail, as expected benchmarks such as `2DConvolution`, `3DConvolution`, `Atax`, `Fdtd2d`, and `Mvt` get better performance rates using the CUDA implementation, while `Covariance`, `Syr2k`, and `Syrk` for the SYCL

Table 4 Average speedup obtained from Polybench suit in the Jetson Orin Nano

Speedup	CUDA	AdaptiveCpp	DPC++	AdaptiveCpp (ARM)
CUDA	1	1.17	1.22	5.75
AdaptiveCpp	–	1	1.07	5.46
DPC++	–	–	1	4.99
AdaptiveCpp (ARM)	–	–	–	1

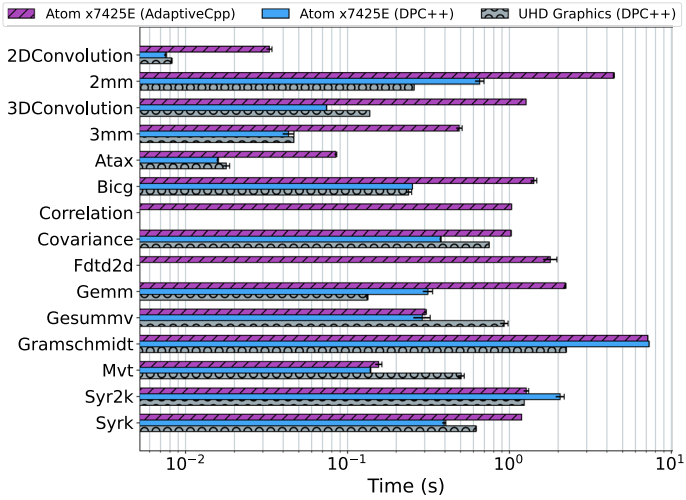

Fig. 3 Execution time recorded for Polybench suite tests on the UP Squared Pro 7000 Edge

version. Furthermore, other benchmarks such as `2mm`, `3mm`, `Bicg`, `Correlation`, `Gemm`, `Gesummv`, or `Gramschmidt` achieved almost equivalent execution time using different programming models.

Table 4 displays the average performance improvement achieved by each compiler. For example, the benchmarks based on CUDA are on average 1.17× faster in comparison with their AdaptiveCpp counterpart. In line with expectations, the CUDA version outperforms the SYCL versions in all cases, achieving an average speedup of 1.17× and 1.22× compared to AdaptiveCpp and DPC++. Shifting our focus to the SYCL implementations, AdaptiveCpp and DPC++ exhibit similar performance metrics. Even in cases where they diverge, differences are small enough to be encompassed by the standard deviation. Consequently, the disparities between the versions are not relevant. Although we have also included the execution on ARM A78AE CPU by means of AdaptiveCpp implementation, it is worth noting that the performance is not particularly favorable when compared to GPU times.

Focusing on the UP Squared Pro 7000 Edge board, Fig. 3 depicts the execution times. It is noteworthy to mention that the Intel UHD GPU could not be utilized

Table 5 Average speedup obtained from Polybench suit in the UP Squared Pro 7000 Edge

Speedup	AdaptiveCpp (Atom)	DPC++(Atom)	DPC++(UHD)
AdaptiveCpp (Atom)	1	0.55	0.79
DPC++ (Atom)	–	1	1.35
DPC++ (UHD)	–	–	1

Table 6 Frames Per Second (FPS) achieved during the execution of optic flow algorithms on various datasets and devices. The table shows the median and the standard deviation of each measure

Dataset	Device	Lucas–Kanade	Horn–Schunck	TV-L[1]
Schoolgirls (432×240)	Ampere GPU (CUDA)	$x \sim =$ 458 FPS $\sigma_X = 119.4$	$x \sim =$ 19.8 FPS $\sigma_X = 0.13$	$x \sim =$ **36.5 FPS** $\sigma_X =$ **0.44**
	Ampere GPU (DPC++)	$x \sim =$ **583 FPS** $\sigma_X =$**49.1**	$x \sim =$ **21.8 FPS** $\sigma_X =$ **0.11**	$x \sim =$ 31.2 FPS $\sigma_X = 0.37$
	UHD Graphics (DPC++)	$x \sim =$ 528 FPS $\sigma_X = 46.9$	$x \sim =$ 16.24 FPS $\sigma_X = 0.19$	$x \sim =$ 12.1 FPS $\sigma_X = 0.15$
Middlebury (640×480)	Ampere GPU (CUDA)	$x \sim =$ 168 FPS $\sigma_X = 2.42$	$x \sim =$ **8.92 FPS** $\sigma_X =$ **0.10**	$x \sim =$ **19.2 FPS** $\sigma_X =$ **0.47**
	Ampere GPU (DPC++)	$x \sim = =$ 147 FPS $\sigma_X = 9.44$	$x \sim =$ 8.66 FPS $\sigma_X = 0.10$	$x \sim =$ 15.17 FPS $\sigma_X = 0.37$
	UHD Graphics (DPC++)	$x \sim =$ **250 FPS** $\sigma_X =$ **2.79**	$x \sim =$ 5.01 FPS $\sigma_X = 0.08$	$x \sim =$ 7.61 FPS $\sigma_X = 0.32$
MPI-Sintel (1024×436)	Ampere GPU (CUDA)	$x \sim =$ 136 FPS $\sigma_X = 1.27$	$x \sim =$ 6.36 FPS $\sigma_X = 0.03$	$x \sim =$ **14.6 FPS** $\sigma_X =$ **0.12**
	Ampere GPU (DPC++)	$x \sim =$ 150 FPS $\sigma_X = 6.61$	$x \sim =$ **6.98 FPS** $\sigma_X =$ **0.07**	$x \sim =$ 12.8 FPS $\sigma_X = 0.1$
	UHD Graphics (DPC++)	$x \sim =$ **214 FPS** $\sigma_X =$ **2.42**	$x \sim =$ 2.98 FPS $\sigma_X = 0.03$	$x \sim =$ 6.09 FPS $\sigma_X = 0.17$

in conjunction with AdaptiveCpp due to the absence of OpenCL or native bare-metal support. Moreover, the `Correlation` and `Fdtd2d` could not run on the DPC++ implementation due to the requirement for double-precision computations. This issue is motivated by the lack of hardware support for double precision on Intel UHD GPU, and for the Atom CPU, the reason is associated to the current OpenCL driver which does not provide support for double precision.

Table 5 summarizes the speedups obtained by each device and SYCL implementation. The Atom CPU with the AdaptiveCpp compiler obtains the worst performance because SYCL code is translated to OpenMP, while DPC++ is conducted by the OpenCL backend. This point makes the difference between both implementations [31, 50]. When comparing DPC++ performance on both CPU and GPU (UHD Graphics), it is noteworthy that the Atom processor even outperforms the GPU. Given the utilization of default-sized problem parameters, it does not appear to be worthwhile to use the GPU.

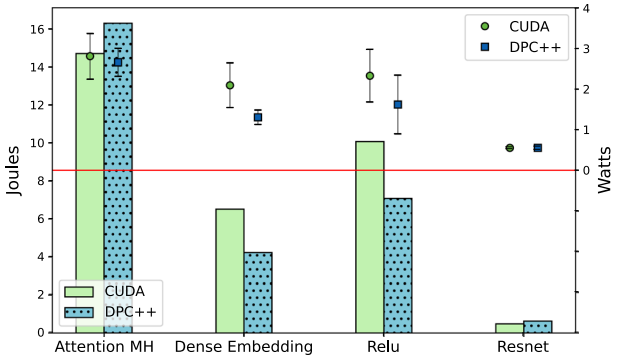

Fig. 4 Jetson Orin Nano's GPU power and energy consumption by language. The bars represent energy consumption in Joules, while the scatter plot illustrates the average power consumption in watts

5.2 Optic flow experiments

Table 6 collects the performance (measured in Frame Per Second-FPS) achieved by varying video resolution, GPU device, accelerator implementation, and algorithm. We have also highlighted in bold type the best result fulfilled in each dataset and algorithm. In order to avoid execution variability test was run 10 times, so the table shows the average and standard deviation. We would like to clarify that we decided to omit the results on CPU devices since either the ARM Cortex-A78E or the Intel Atom X7425E are far away from the GPU counterpart execution times. Furthermore, for the sake of clarity, we have also removed from the final results AdaptiveCpp, due to the similar times achieved with DPC++.

Regarding the LK algorithm, it is noteworthy that the Intel UHD Graphics is the most suitable device when resolution increases. For the HS algorithm, the Ampere GPU is prominent, but distinguishing between CUDA and DPC++ implementations in terms of performance is challenging, as in most instances, both implementations yield nearly identical fps. Using the TV-L^1 algorithm as a benchmark, once again, we observe that the Ampere GPU reports the best performance rates. Diving deeper into the comparison of implementations on the Ampere GPU, an average difference of approximately 2.9% is observed between CUDA and DPC++. When examining each algorithm individually, we find that LK exhibits a 5.4% improvement with DPC++, HS favors DPC++ by 5%, and the TV-L^1 implementation performs 19% better with CUDA.

On the UHD Graphics side, a direct comparison is made with the Ampere GPU along DPC++. The overall difference is 60.9% in favor of the Orin GPU. When looking at each algorithm individually, we observe a 20.2% improvement for the UHD Graphics in the LK algorithm, an 80.4% advantage for the Ampere GPU in the HS algorithm, and 122% for the Ampere GPU in the TV-L^1 algorithm.

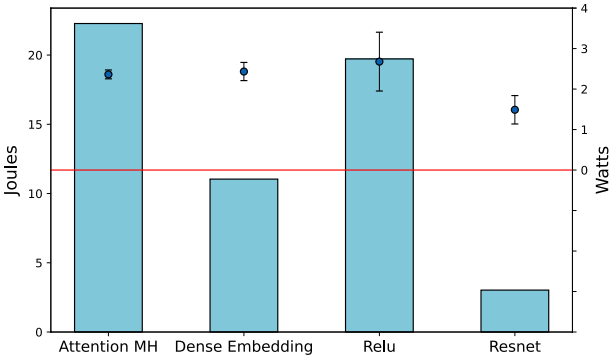

Fig. 5 UP Squared Pro 7000 Edge's GPU power and energy consumption in DPC++. The bars represent energy consumption in Joules, while the scatter plot illustrates the average power consumption in watts

5.3 Energy consumption results

The following subsection attempts to address the energy and power consumption associated with each language and board. In this case, we have transitioned to AI typical benchmarks widely used in the edge area.

Fig. 4 depicts to Ampere GPU power and energy consumption. The bars represent the average energy consumption by each benchmark, while the scatter plot represents the average and standard deviation of the power consumption. At first glance, SYCL reduces the energy consumption in `Dense Embedding` and `Relu` benchmarks, while CUDA does in `Attention Multi-Head` and `Resnet` tests. In those that SYCL beats, we found that the overall consumption was less than CUDA's, but not the time performance. `Dense Embedding` gives *3.11s vs 3.24s*, and `Relu` through *4.32s vs 4.37s* for CUDA and SYCL, respectively. However, the average power consumption was *2.1 watts vs 1.31 watts* for the `Dense Embedding` test, while in the `Resnet` was *2.33 watts vs 1.62 watts*. Hence, the overall energy consumption makes SYCL less energy consumption than CUDA.

Across the mentioned benchmarks, the total energy consumption in CUDA was 31.8J, whereas in SYCL, it was 28.3J. These numbers indicate that SYCL is 12% more energy-efficient than CUDA. The CUDA binary exhibits power over-consumption to enhance performance, but this increase does not proportionally scale with the overall energy consumption.

On the other hand, Fig. 5 illustrates the power and energy consumption of the Intel GPU. In this instance, we only measured the DPC++ implementation due to the aforementioned issues with AdaptiveCpp. The overall consumption was 56J. A direct comparison with the NVIDIA board is not feasible due to the differences in measurement tools. While Jetson's *tegrastats* provides the watt consumption of the combined CPU+GPU package, the Squared Pro's *turbostat* disaggregates the CPU and GPU consumption. When we aggregate the CPU and GPU power consumption, the overall energy consumption increases to 194J. This is 6.33 times higher than the consumption of the Jetson SoC. It is worth noting that, while the Jetson features an

Arm CPU, known for its energy efficiency, the UP Squared employs an Intel Atom, an x86-64 architecture which is generally more energy-demanding [51].

6 Discussion

SYCL showed the benefits of using as programming model in the edge market segment. In fact, we could successfully run the same code on different devices without an important performance degradation: Two edge boards from different vendors were employed for the same task.

In the initial phase, we tested SYCL along with the aforementioned boards using the Polybench suite. This part of the experiment aimed to demonstrate the ability to run the same SYCL code on various architectures, the differences between the most commonly used SYCL implementations, and the minimal performance differences when compared to native implementations, such as CUDA. Polybench results help shed light on these objectives. First and foremost, thanks to SYCL, the Polybench suite was able to run on x86/64 CPU, ARM CPU, NVIDIA GPU, and Intel GPU. This is the primary advantage of SYCL when compared to native implementations, as it simplifies development across various architectures. It is evident that employing the SYCL language to articulate an application's parallelism not only ensures portability across various architectures and vendors but also enhances productivity.

On one hand, it is also important to mention that regarding to DPC++ and AdaptiveCpp, both encountered difficulties in running on all the architectures tested. DPC++ failed to operate on ARM CPUs, while AdaptiveCpp encountered issues with Intel GPUs. Nonetheless, these problems could be addressed through improved documentation on how to compile AdaptiveCpp for Intel GPUs using OpenCL or Level0 backends, or by employing open-source OpenCL implementations for ARM CPUs such as *pocl*.[17] Regarding their performance, the CUDA GPU architectures exhibited minimal variation, approximately 7%, which depended on the specific benchmark being observed. Conversely, on x86/64 CPUs, AdaptiveCpp failed to achieve comparable results to DPC++ with a notable 45% drop in performance based on the underlying OpenMP conversion for CPU architectures. However, it is important to note that OpenMP compatibility can be advantageous for emerging architectures like the promising RISC-V.

On the other hand, it is important to consider the comparison between SYCL implementations and CUDA. The overall metric indicates that CUDA outperforms SYCL by approximately 17-22%, depending on the specific implementation. Nevertheless, when examined on a benchmark-by-benchmark basis, the superiority of CUDA is not consistently clear-cut. Out of the five tests performed better with CUDA, while SYCL excelled in the other three, and the remaining eight showed similar performance. In light of these results, it can be inferred that utilizing SYCL does not significantly degrade performance at all.

In a subsequent analysis, we utilized an AI subset comprising four benchmarks of common algorithms widely employed in edge computing. The objective of this

[17] https://github.com/pocl/pocl

Table 7 USD per minute and frame computed by the GPUs and datasets

		Lucas–Kanade	Horn–Schunck	TV-L[1]
Shoolgirls (432 x 240)	Ampere GPU	$0.01 per frame	$**0.38 per frame**	$**0.27 per frame**
	UHD Graphics	$**0.01 per frame**	$0.41 per frame	$0.55 per frame
Basketball (640 x 480)	Ampere GPU	$0.06 per frame	$**0.96 per frame**	$**0.55 per frame**
	UHD Graphics	$**0.03 per frame**	$1.33 per frame	$0.87 per frame
MPI-Sintel (1024 x 436)	Ampere GPU	$0.06 per frame	$**1.19 per frame**	$**0.64 per frame**
	UHD Graphics	$**0.03 per frame**	$2.23 per frame	$1.09 per frame

assessment was to evaluate the energy consumption when transitioning from CUDA to SYCL and to compare the energy efficiency between boards. The initial analysis indicated that SYCL exhibited lower performance in those benchmarks, but it was also less power-demanding, resulting in overall lower energy consumption compared to CUDA (approximately 12% less). However, we warn the reader that the energy efficiency observed depends on the specific application and workload tested. Since other algorithms and workloads tested change observed behavior, take as a rule of thumb that approximately $\pm$10% of the energy consumption would vary by moving from CUDA to SYCL and how well the application was tuned.

The other phase of the experiment aimed to test SYCL in real-world scenarios. One common application in the edge computing sphere is computer vision, with a specific focus on optical flow in this case. We evaluated three different datasets-Schoolgirls, Middlebury, and MPI-Sintel-using three optical flow algorithms: LK, HS, and TV-L[1]. There are two points that need to be addressed: the performance difference between CUDA-SYCL and the SYCL comparison between boards.

The primary distinction among the implementations is evident in the TV-L[1] algorithm. HS demonstrates similar processing times in both versions, while LK stands out as an exceptionally lightweight algorithm worth considering. The variances in TV-L[1] processing times—16% for Schoolgirls, 26% for Middlebury, and 14% for MPI-Sintel—should be viewed as the price for achieving code portability. Maintaining different versions of the same algorithm, even if it outperforms, inevitably raises development costs. Therefore, SYCL for edge computing, like in other platforms such as HPC, is no exception and also incurs a "minor" cost of 19% to ensure portability.

Lastly, employing SYCL could also be a sensible choice when comparing across various boards. This is because it helps narrow the gap between the software and the algorithm's implementation, which can vary depending on the programming language used. To demonstrate this premise, the theoretical performance of the Jetson Orin Nano GPU is 1,280 GFLOPS; meanwhile, the UP Squared Pro 7000 Edge GPU offers 460 GFLOPS (a 178% difference). The respective board prices are $499 and $399 (a 25% variance). Therefore, given the performance results and the actual performance achieved in optical flow, a comparison based on cost is warranted. Table 7 presents an assessment of the monetary cost (in dollars) per performance unit (FPS). SYCL is used to conduct the results.

Table 8 Energy consumed by frame processed in DPC++ and multiple optic flow algorithms. MPI-Sintel dataset was used to conduct the results

	Lucas–Kanade	Horn–Schunck	TV-L[1]
Jetson Orin Nano	**5.59 mJ per frame**	**414 mJ per frame**	**231 mJ per frame**
UP Squared	8.42 mJ per frame	1,599 mJ per frame	1,041 mJ per frame

When analyzing the algorithms, we observed the following cost differences: For LK, the UHD GPU is 4.5% less expensive, while for HS is 27% more cost-effective than the Orin GPU. In the case of TV-L[1], the Orin board's cost is 43% lower.

On the consumption side, Table 8 presents the millijoules (mJ) consumed by each processed frame. The table results depict the most demanding dataset, the MPI-Sintel. It is important to note that energy consumption measures not only the GPU power but also that of the SoC, which includes the CPU. Remarkably, the Jetson board demonstrates significantly greater energy efficiency compared to the UP Squared, with differences ranging from 1.5 to 4.5 times greater energy efficiency. The main issue with the UP Squared is that its CPU consumes more power in idle states than the Arm's CPU.

These comparative analyses allow us to select the most suitable board based on specific priorities like cost, power consumption, performance, or real-time demands. The ability to use a single, portable code greatly enhances decision-making efficiency and promotes the widespread deployment of IoT applications on various architectures and vendors, eliminating the need for maintaining multiple development efforts or dependency on the commercial policies of a specific manufacturer.

7 Conclusion

The rapid growth of edge computing has introduced various solutions, many of which incorporate low-power accelerators to enhance performance. Accelerators are typically designed to work with specific custom languages such as CUDA, HIP, VHDL, and others. However, this approach creates compatibility issues, as it necessitates customizing the code for each architecture.

This work demonstrated the ability of edge computing to execute and leverage SYCL code on different boards and custom accelerators. We employed the Polybench suite to evaluate various SYCL implementations on the same hardware, and the performance gap was found to be negligible. On the energy side, we utilized a HeCBench subset, and no differences were observed.

Additionally, we utilized a realistic computer vision application based on optical flow algorithms to assess the practical application of SYCL in edge computing scenarios. The experiments revealed a performance disparity between native solutions like CUDA and SYCL. Nevertheless, we deliberated on the significance of SYCL's portability in development tasks and the trade-off in performance that developers may encounter. Utilizing a single, portable code streamlines decision-making and

enables broad IoT deployment across different architectures and vendors, reducing the reliance on multiple development efforts and specific manufacturer policies. To the best of the author's knowledge, this work represents one of the earliest efforts focused on edge computing and code portability utilizing SYCL.

Future work should focus on incorporating performance portability metrics to facilitate a comparison with the native version. Given the prevalent use of edge computing in image processing and real-time applications, further investigations could explore the advantages of employing SYCL in image processing frameworks such as OpenCV. Moreover, extending the research to encompass other edge devices and evaluating their performance and power consumption would provide valuable insights.

Author Contributions All authors contributed to the research in the main concepts and design. The software was developed by Y. FR. Y. FR also performed experiments. C. G. analyzed the results and proposed methodology in the experimentation phase. All authors write and approve the final manuscript.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This paper has been supported by the EU (FEDER), the Spanish MINECO under grants PID2021-126576NB-I00 and TED2021-130123B-I00 funded by MCIN/AEI/10.13039/501100011033 and by European Union "ERDF A way of making Europe" and the NextGenerationEU/PRT.

Data availability statement Some datasets employed for the current study are available in the *artecs-group/sycl-optic-flow* repository, https://github.com/artecs-group/sycl-optic-flow/tree/main/dataset. The full datasets can be found in [43].

Code availability The code supporting the results of this article is available in the *artecs-group/sycl-optic-flow* repository, https://github.com/artecs-group/sycl-optic-flow

Declarations

Conflict of interest Do not have any conflicts of interest with your journal and no mutual conflicts of interest among the authors.

Ethical approval Not applicable for this item.

References

1. Cao K, Liu Y, Meng G, Sun Q (2020) An overview on edge computing research. IEEE Access 8:85714–85728. https://doi.org/10.1109/ACCESS.2020.2991734

2. Mansouri Y, Babar MA (2021) A review of edge computing: features and resource virtualization. J Parallel Distribut Comput 150:155–183. https://doi.org/10.1016/j.jpdc.2020.12.015
3. Satyanarayanan M (2017) The emergence of edge computing. Computer 50(1):30–39. https://doi.org/10.1109/MC.2017.9
4. Kong X, Wu Y, Wang H, Xia F (2022) Edge computing for internet of everything: a survey. IEEE Int Things J 9(23):23472–23485. https://doi.org/10.1109/JIOT.2022.3200431
5. Tripathy B, Anuradha J (2018) Internet of Things (IoT): Technologies, Applications, Challenges and Solutions, p. 358. CRC press, USA. https://www.routledge.com/Internet-of-Things-IoT-Technologies-Applications-Challenges-and-Solutions/Tripathy-Anuradha/p/book/9780367572921
6. Afzal B, Umair M, Shah GA, Ahmed E (2019) Enabling iot platforms for social iot applications: vision, feature mapping, and challenges. Future Gener Comput Syst 92:718–731
7. Tavana M, Hajipour V, Oveisi S (2020) Iot-based enterprise resource planning: Challenges, open issues, applications, architecture, and future research directions. Internet of Things 11:100262
8. Himeur Y, Alsalemi A, Al-Kababji A, Bensaali F, Amira A, Sardianos C, Dimitrakopoulos G, Varlamis I (2021) A survey of recommender systems for energy efficiency in buildings: principles, challenges and prospects. Inf Fusion 72:1–21. https://doi.org/10.1016/j.inffus.2021.02.002
9. Ramachandran P, Ranganath S, Bhandaru MK, Tibrewala S (2021) A survey of ai enabled edge computing for future networks. In: 2021 IEEE 4th 5G World Forum (5GWF), 459–463
10. Intel: oneAPI DPC++ Compiler and Runtime architecture design. https://intel.github.io/llvm-docs/design/CompilerAndRuntimeDesign.html (2023)
11. Keryell R, Reyes R, Howes L (2015) Khronos sycl for opencl: a tutorial. In: Proceedings of the 3rd International Workshop on OpenCL, pp. 1–1
12. Buck I (2007) Gpu computing with nvidia cuda. In: ACM SIGGRAPH 2007 Courses, p. 6
13. Bauman P, Chalmers N, Curtis N, Freitag C, Greathouse J, Malaya N, McDougall D, Moe S, van Oostrum R, Wolfe N, et al (2019) Introduction to amd gpu programming with hip. Presentation at Oak Ridge National Laboratory. Online at: https://www. olcf. ornl. gov/calendar/intro-to-amd-gpu-programming-with-hip
14. Reinders J, Ashbaugh B, Brodman J, Kinsner M, Pennycook J, Tian X (2023) Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL. Second Edition, Springer, USA. https://doi.org/10.1007/978-1-4842-9691-2
15. Castaño G, Faqir-Rhazoui Y, García C, Prieto-Matías M (2022) Evaluation of intel's dpc++ compatibility tool in heterogeneous computing. J Parallel Distribut Comput 165:120–129. https://doi.org/10.1016/j.jpdc.2022.03.017
16. Deakin T, McIntosh-Smith S (2020) Evaluating the performance of hpc-style sycl applications. In: Proceedings of the International Workshop on OpenCL. IWOCL '20. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3388333.3388643
17. Breyer M, Van Craen A, Pflüger D (2022) A comparison of sycl, opencl, cuda, and openmp for massively parallel support vector machine classification on multi-vendor hardware. In: International Workshop on OpenCL. IWOCL'22. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3529538.3529980
18. Kang P (2023) Programming for high-performance computing on edge accelerators. Mathematics. https://doi.org/10.3390/math11041055
19. Angus D, Georgiev S, Arroyo Gonzalez H, Riordan J, Keir P, Goli M (2023) Porting sycl accelerated neural network frameworks to edge devices. In: Proceedings of the 2023 International Workshop on OpenCL. IWOCL '23. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3585341.3585346
20. Khronos SYCL working group: SYCL Specification. https://registry.khronos.org/SYCL/ (2023)
21. OpenMP: The OpenMP Specification. https://www.openmp.org/ (2023)
22. Khronos SYCL working group: The OpenCL Specification. https://registry.khronos.org/OpenCL/ (2023)
23. Ludwig K (2021) Performance portability and evaluation of heterogeneous components of seissol targeted to upcoming intel hpc gpus
24. LLVM-Project: User Guide for AMDGPU Backend. https://www.llvm.org/docs/AMDGPUUsage.html (2023)

25. Marangoni M, Wischgoll T (2016) Togpu: automatic source transformation from C++ to cuda using clang/llvm. Electron Imag 2016(1):1–9
26. illuhad (2021) AdaptiveCpp design and architecture. https://github.com/OpenSYCL/OpenSYCL/blob/develop/doc/architecture.md
27. Jin Z (2020) The rodinia benchmark suite in sycl. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States). Argonne Leadership ..
28. Tramm JR, Siegel AR, Islam T, Schulz M (2014) Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis. The Role of Reactor Physics toward a Sustainable Future (PHYSOR)
29. 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, NY, USA. https://doi.org/10.1145/3529538.3530005
30. Grauer-Gray S, Xu L, Searles R, Ayalasomayajula S, Cavazos J (2012) Auto-tuning a high-level language targeted to gpu codes. In: 2012 Innovative Parallel Computing (InPar), pp. 1–10. https://doi.org/10.1109/InPar.2012.6339595
31. Lal S, Alpay A, Salzmann P, Cosenza B, Hirsch A, Stawinoga N, Thoman P, Fahringer T, Heuveline V (2020) Sycl-bench: a versatile cross-platform benchmark suite for heterogeneous computing. In: Euro-Par 2020: Parallel Processing: 26th International Conference on Parallel and Distributed Computing, Warsaw, Poland, August 24–28, 2020, Proceedings 26, pp. 629–644. https://doi.org/10.1007/978-3-030-57675-2_39. Springer
32. Stiller C, Konrad J (1999) Estimating motion in image sequences. IEEE Signal Process Mag 16(4):70–91. https://doi.org/10.1109/79.774934
33. Baker S, Roth S, Scharstein D, Black MJ, Lewis JP, Szeliski R (2007) A database and evaluation methodology for optical flow. In: 2007 IEEE 11th International Conference on Computer Vision, pp. 1–8. https://doi.org/10.1109/ICCV.2007.4408903
34. Butler DJ, Wulff J, Stanley GB, Black MJ (2012) A naturalistic open source movie for optical flow evaluation. In: Computer Vision–ECCV 2012: 12th European Conference on Computer Vision, Florence, Italy, October 7-13, 2012, Proceedings, Part VI 12, pp. 611–625. Springer
35. Geiger A, Lenz P, Stiller C, Urtasun R (2013) Vision meets robotics: the KITTI dataset. Int J Robot Res 32(11):1231–1237. https://doi.org/10.1177/0278364913491297
36. Horn BKP, Schunck BG (1981) Determining optical flow. Artif Int 17(1):185–203. https://doi.org/10.1016/0004-3702(81)90024-2
37. Sun D, Roth S, Black MJ (2010) Secrets of optical flow estimation and their principles. In: 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp. 2432–2439. https://doi.org/10.1109/CVPR.2010.5539939
38. Borzì A, Schulz V (2009) Multigrid methods for PDE optimization. SIAM Rev 51(2):361–395. https://doi.org/10.1137/060671590
39. Lucas BD, Kanade T (1981) An iterative image registration technique with an application to stereo vision. In: Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2. IJCAI'81, pp. 674–679. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA
40. Botella G, Garcia A, Rodriguez-Alvarez M, Ros E, Meyer-Baese U, Molina MC (2010) Robust bioinspired architecture for optical-flow computation. IEEE Trans Very Large Scale Integrat VLSI Syst 18:616–629
41. Gong Y, Zhang J, Liu X, Li J, Lei Y, Zhang Z, Yang C, Geng L (2023) A real-time and efficient optical flow tracking accelerator on fpga platform. In: IEEE Transactions on Circuits and Systems I: Regular Papers, 1–14. https://doi.org/10.1109/TCSI.2023.3298969
42. Jaiswal D, Kumar P (2022) A survey on parallel computing for traditional computer vision. Concurr Comput : Pract Exp 34(4):6638
43. Zhai M, Xiang X, Lv N, Kong X (2021) Optical flow and scene flow estimation: a survey. Pattern Recog 114:107861. https://doi.org/10.1016/j.patcog.2021.107861
44. Zach C, Pock T, Bischof H (2007) A duality based approach for realtime tv-l1 optical flow. In: Proceedings of the 29th DAGM Conference on Pattern Recognition, Springer, Berlin, Heidelberg
45. Wedel A, Pock T, Zach C, Bischof H, Cremers D (2009) An improved algorithm for tv-l1 optical flow. In: Statistical and Geometrical Approaches to Visual Motion Analysis: International Dagstuhl Seminar, Dagstuhl Castle, Germany, July 13-18, 2008. Revised Papers, Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-03061-1_2
46. Sánchez Pérez J, Meinhardt-Llopis E, Facciolo G (2013) TV-L1 optical flow estimation. Image Process On Line 3:137–150. https://doi.org/10.5201/ipol.2013.26

47. Romera T, Petreto A, Lemaitre F, Bouyer M, Meunier Q, Lacassagne L, Etiemble D (2023) Optical flow algorithms optimized for speed, energy and accuracy on embedded Qpus. J Real-Time Image Process 20(2):32. https://doi.org/10.1007/s11554-023-01288-6
48. Romera T, Petreto A, Lemaitre F, Bouyer M, Meunier Q, Lacassagne L (2021) Implementations impact on iterative image processing for embedded gpu. In: 2021 29th European Signal Processing Conference (EUSIPCO), pp. 736–740. https://doi.org/10.23919/EUSIPCO54536.2021.9615947
49. Alpay A, Heuveline V (2020) Sycl beyond opencl: The architecture, current state and future direction of hipsycl. In: Proceedings of the International Workshop on OpenCL. IWOCL '20. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3388333.3388658
50. Alpay A, hipSYCL 0.9.2 - compiler-accelerated CPU backend, nvc++ support and more. https://adaptivecpp.github.io/hipsycl/release/cpu/extension/nvc++/hipsycl-0.9.2/
51. Jarus M, Varrette S, Oleksiak A, Bouvry P (2013) Performance evaluation and energy efficiency of high-density HPC platforms based on intel, Amd and arm processors. In: Pierson J-M, Da Costa G, Dittmann L (eds) Energy Eff Large Scale Distribut Syst. Springer, Berlin, Heidelberg, pp 182–200