The Journal of Supercomputing (2024) 80:13295–13316
https://doi.org/10.1007/s11227-024-05958-5

Assessing Intel OneAPI capabilities and cloud-performance for heterogeneous computing

Silvia R. Alcaraz[1] · Ruben Laso[1,3] · Oscar G. Lorenzo[1,2] · David L. Vilariño[2] · Tomás F. Pena[1,2] · Francisco F. Rivera[1,2]

Accepted: 3 February 2024 / Published online: 4 March 2024

Abstract

This work presents a performance-oriented study of a heterogeneous application developed with Intel OneAPI to solve two well-known diffusion problems: heat diffusion and image denoising. We have explored CPU+iGPU and CPU+FPGA schemes, applying dynamic load balancing and conducting experiments on Intel DevCloud. The results demonstrate that the CPU+iGPU scheme outperforms the execution times achieved by the fastest device when the problem is sufficiently computationally demanding. We also found that the performance of the CPU+FPGA scheme is heavily affected by bandwidth limitations and specific strategies to manage memory efficiently are required. Moreover, it was demonstrated that dynamic workload balancing is crucial due to possible performance fluctuations in any of the implicated devices. In conclusion, Intel OneAPI provides a helpful tool for multi-platform development using a unique high-level language, DPC++. However, developing specific code for each platform is necessary to achieve optimal performance.

Keywords Intel OneAPI · Intel DevCloud · GPU · FPGA · Heterogeneous computing · Cloud computing

✉ Silvia R. Alcaraz
silvia.alcaraz@usc.es

1 Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela, Rúa de Jenaro de la Fuente Domínguez S/N, 15782 Santiago de Compostela, Galicia, Spain

2 Departamento de Electrónica e Computación, Universidade de Santiago de Compostela, Rúa Lope Gómez de Marzoa, S/N, 15782 Santiago de Compostela, Galicia, Spain

3 Faculty of Informatics, Research Group for Parallel Computing, TU Wien, Treitlstraße 3, 1040 Vienna, Austria

1 Introduction

Technological advances during the last few years have caused computer applications to become increasingly computationally demanding. This fact not only affects applications used in the scientific field but also includes applications we use on a daily basis. Simultaneously, as applications or programs demand more computing power, manufacturers have also been producing more capable devices. Nowadays, mobile phones are more powerful in memory and processing than desktop computers that existed at the beginning of the century. This evolution can be traced to the rise in powerful processors and larger memory capacities. Moreover, there has been a growth of devices harnessing the advantages offered by heterogeneous computing. This paradigm refers to utilising multiple processing units or accelerators within a unified computing system, which may have different architectures or capabilities. It aims to leverage the strengths inherent in each of these devices to attain a range of advantages, including enhanced performance and reduced energy consumption, among others.

Personal computers or laptops are usually equipped with a CPU and an integrated and/or dedicated GPU. This is related to the GPGPU (General-Purpose Computing on Graphics Processing Units) concept, which is based on leveraging the computing power of GPUs for general-purpose applications, not just for rendering graphics. Nowadays, the most popular platforms to develop GPGPU applications are CUDA (Compute Unified Device Architecture) [1], to work with NVIDIA GPUs, and OpenCL (Open Computing Language) [2]. The last is an open framework that can run on various devices, including CPUs, GPUs, and other accelerators.

Note that GPUs are not the only devices to be included to create heterogeneous schemes, although they are the most widely used because of the ecosystem of high-level tools to develop applications for them. Nevertheless, it is relevant to emphasise that other devices exist, such as FPGAs (Field Programmable Gate Arrays). In the past, the use of FPGAs increased due to their capacity to adapt their architecture to a specific problem based on programmable logic. Another advantage is their low power consumption, as demonstrated in several studies. Regarding this, Betkaoui et al. [3] determined the advantages of FPGA-based HPC (High Performance Computing) systems over GPUs regarding energy efficiency. Cong et al. [4] also described some performance differences between FPGAs and GPUs, where the results show that FPGAs used less power than GPUs. To make this comparison, they used the well-known benchmark for GPUs, Rodinia [5], migrating some kernels with Vivado HLS (High-Level Synthesis) C [6] to be executed on FPGAs. Both studies highlight the low power consumption of FPGAs, although they point out the limitations that bandwidth or memory management may cause in these devices.

Despite the advantages mentioned about FPGAs, their popularity did not reach the magnitude of GPUs mainly because of the maturity in the software stack provided by the latter. The standard way to program FPGAs requires using hardware description languages such as VHDL or Verilog and low-level knowledge of hardware and circuitry to develop optimal solutions. However, FPGAs are currently experiencing a new resurgence thanks to the new high-level development tools [7]. One of the challenges faced by such tools is to provide a compromise between programming simplicity and performance portability. The concept of performance portability refers to the capacity of a code to be executed efficiently across different platforms without the necessity for significant manual modifications. Strategies for parallelising a problem or optimising its performance are different across devices. To develop efficient codes for GPUs, it is essential to understand the thread execution model to maximise the utilisation of their computing capabilities. Additionally, achieving coalescent accesses and allocating data efficiently is equally important to best use the memory. For dGPUs (dedicated GPUs), careful consideration is necessary when deciding which data to send to on-chip memory to avoid costly memory communications. For iGPUs (integrated GPUs), having some memory shared with the CPU can save data transfers, although this type of GPUs tends to have lower computational power. FPGAs are a completely different device; therefore, other techniques are used to optimise codes. Task decomposition and pipelining are essential, enabling the exploitation of parallel processing inherent in FPGAs. Furthermore, it is necessary to use the on-chip memory efficiently, coupled with a selection of appropriate memory access patterns, to minimise data movement and reduce latency.

In the case of Intel, the specific tool Intel OneAPI [8] includes a software package that allows the creation of high-level, cross-platform solutions using a single programming language, DPC++ (Data Parallel C++) [9]. It offers a syntax based on C++17 and follows the heterogeneous programming model proposed by SYCL [10] to provide multi-platform programming mechanisms. The leading platforms included are CPUs, GPUs, and FPGAs, although there is also support for custom platforms. Additionally, it is possible to use this tool remotely on Intel Developer Cloud—or Intel DevCloud—where several models of CPUs, GPUs, and FPGAs manufactured by Intel are hosted.

This work presents a performance-oriented study of two heterogeneous schemes using well-known iterative diffusion problems as case studies, specifically heat diffusion and image denoising. In addition, dynamic workload balancing is employed to automate the optimal distribution between the involved devices without manually tuning the parameters. The tool used for the development has been Intel OneAPI, as it allows the creation of cross-platform code for CPUs, GPUs and FPGAs. Thus, we have successfully assessed and compared the performance of the CPU+iGPU and CPU+FPGA schemes in the Intel DevCloud environment. Likewise, we have examined the tool's usability and assessed the performance attained on each device by implementing a generic kernel code applicable to all target platforms. In summary, the main two contributions of this work are the evaluation of the Intel OneAPI tool in terms of usability, performance portability and throughput, particularly, in the Intel DevCloud environment; and a comparative performance study between two heterogeneous computing schemes, CPU+iGPU and CPU+FPGA, using dynamic load balancing.

The rest of the paper is structured as follows: Sect. 2 includes an overview of the related work; Sect. 3 introduces the case studies utilised in this work, namely, the heat diffusion and image denoising problems; Sect. 4 discusses the implementations of the benchmarks; The experimental environment is detailed in Sect. 5; Results are shown and discussed in Sect. 6; Finally, conclusions and future work are outlined in Sect. 7.

2 Related work

Heterogeneous computing offers a model that allows further progress in achieving greater computing power, energy efficiency and optimised code development to address specific tasks. However, not all problems can be effectively leveraged using heterogeneous computing schemes. Recommended prerequisites include the possibility to decompose them into independent or loosely coupled tasks and the capacity to avoid data dependencies, ensuring that each device can process its amount of data without conflicts.

Regarding the applications that can be improved using heterogeneous platforms, Lukarski and Neytcheva [11] studied the particular case of iterative methods, determining that the codes to solve this type of problem can achieve better performance using parallel and heterogeneous approaches. Numerous studies in the literature related to this topic further reinforce this claim. Among them, Venkatasubramanian et al. [12] present multi-CPU and multi-GPU implementations of Jacobi's iterative method to solve the 2D Poisson equation. Benner et al. [13, 14] accelerated Newton's iteration for the matrix sign function using hybrid CPU-GPU platforms. Agulleiro et al. [15] proposed a hybrid scheme using multicore CPUs and NVIDIA GPUs to improve the iterative tomographic reconstruction. More recently, Halbiniak et al. [16] presented a port to CPU+GPU for the solidification modelling using OpenCL.

As we explore heat diffusion and image denoising as case studies in this work, it is noteworthy to highlight previous studies that utilised heterogeneous computing to achieve improved performance in addressing these problems. Belhaus et al. [17] conducted a comparative study of parallel heat equation implementations on CPU and NVIDIA GPUs. Sánchez et al. [18] presented an algorithm for removing impulsive noise in images. They used OpenMP (Open Multi-Processing) [19] and CUDA to develop solutions for multi-CPU, multi-GPU and a combination of both. Also, Sarjanoja et al. [20] proposed an implementation of the BM3D (Block-Matching and Three-Dimensional) filtering denoising algorithm for CPU and GPU platforms using OpenCL and CUDA.

Concerning the use of Intel OneAPI in heterogeneous architectures, the literature contains works such as Constantinescu et al.'s [21] implementation of Markov decision processes on low-power CPU+GPU SoCs. Moreover, they compared OpenCL + TBB (Threading Building Blocks) and Intel OneAPI + TBB implementations in terms of ease of programming and performance. Yong et al. [22] migrated an ultrasound imaging application from CUDA to Intel oneAPI, tuning it for GPU, FPGA and CPU. Lupescu and Ţăpuş [23] developed a hybrid hashtable for an SoC (System-On-a-Chip) composed of a CPU and an iGPU. Marinelli et Raja [24] ported a GPU-based hash join algorithm from CUDA to DPC++ and tested it on dGPU, iGPU and multicore CPUs. They also developed OneJoin [25], an edit similarity join between architectures for DNA data storage. Nozal and Bosque [26] assess the performance and energy efficiency of popular HPC benchmarks using dynamic and static workloads on CPU+GPU heterogeneous systems. Najmeh et al. [27] provided HosNa, a benchmark suite written in DPC++ for heterogeneous architectures. In addition, they analysed the performance of CPUs and FPGAs—both manufactured by Intel—using algorithms from the suite. The study by Kashino et al. [28] explores the use of GPU+FPGA for complex physical simulations. Groth et al. [29] presented a hybrid and accelerated implementation of various hash table operations for dGPUs and APUs (Accelerated Processing Units), consisting of a combination of a CPU and an iGPU. Specifically, they used Intel OneAPI to develop the search operation in the case of the APU. Finally, Li et al. [30] propose a portable framework for large-scale graph processing based on the heterogeneous CPU+GPU scheme.

3 Case studies

To study the performance implications of the proposed heterogeneous computing schemes, two diffusion problems have been selected as case studies. This type of problem aims to provide easily understandable examples suitable to be solved using a heterogeneous approach.

3.1 Heat diffusion

As an example of isotropic diffusion, we chose the two-dimensional heat equation:

$$\frac{\partial u}{\partial t} - D\Delta u = 0. \tag{1}$$

The equation states that the rate of change of temperature u with respect to time at a certain point is proportional to the Laplacian of the temperature at that point, which is the sum of the second-order partial derivatives of the temperature with respect to both x and y. In other words, the equation describes how the temperature at a point changes over time due to the temperature differences between its neighbours and the material's thermal properties in the region, as governed by the diffusion coefficient D.

As stated in (1), this problem is an isotropic diffusion because it assumes that heat diffuses uniformly in all directions, regardless of the orientation of the coordinates axes. This is caused by the scalar differential Laplacian operator, which is invariant under rotations of the coordinate axes.

The selected initial conditions for our case study are

$$u(x, y, 0) = \sin(x)\sin(y), \tag{2}$$

where x and y are the coordinates for the x and y directions, respectively, and the temperature at a point (x, y) on the grid at time t is denoted by $u_{x,y,t}$.

Function (2) is a natural choice for the initial temperature distribution as it satisfies the following properties: firstly, it is a smooth function with no sharp edges or corners, so the temperature distribution is continuous and differentiable everywhere in the domain; and secondly, the temperature at the boundary is zero, and this function fulfils that condition, since

$$u(0, y, t) = u(x, 0, t) = u(\pi, y, t) = u(x, \pi, t) = 0. \tag{3}$$

Therefore, the problem can be summarised as follows:

$$\begin{cases} \frac{\partial u}{\partial t} - D\Delta u = 0, \\ u(x, y, 0) = \sin(x)\sin(y), \\ u(0, y, t) = u(x, 0, t) = u(\pi, y, t) = u(x, \pi, t) = 0. \end{cases} \tag{4}$$

3.1.1 Discretisation

To solve the heat equation using finite differences, it is necessary to apply a discretisation of the spatial domain and the time interval. Given the domain $\Omega = (0, \pi) \times (0, \pi)$, it can be discretised in a structured mesh of $N \times N$ nodes spaced by δ_x and δ_y in the x and y axis, respectively such that $\delta_x = \delta_y = \frac{\pi}{N}$.

Regarding time discretisation, we use a fixed time step, δ_t. The problem is solved using an Explicit Euler method, defined as

$$u(x, y, t + \delta_t) = u(x, y, t) + \delta_t \frac{\partial u(x, y, t)}{\partial t}, \tag{5}$$

where u is the temperature.

In this case, we use the well-known approach 5-point scheme [31], shown in Fig. 1. Therefore, it is needed to divide the region of interest into a grid of points with uniform spacing in both x and y directions, being δ_x and δ_y the spacing between two adjacent grid points in the x and y directions.

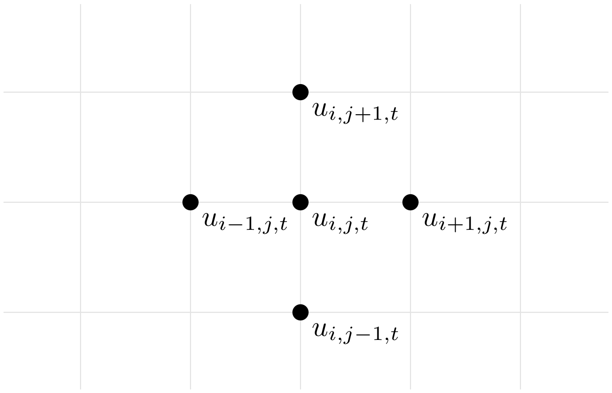

Fig. 1 Laplacian 5 point scheme

The following equations show the approximation of second-order derivatives with respect to x and y using the central difference approximation, respectively:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1,j,t} - 2u_{i,j,t} + u_{i-1,j,t}}{\delta_x^2}, \quad \frac{\partial^2 u}{\partial y^2} \approx \frac{u_{i,j+1,t} - 2u_{i,j,t} + u_{i,j-1,t}}{\delta_y^2}. \tag{6}$$

Finally, substituting these approximations into the original heat equation, we obtain

$$u_{i,j,t+\delta_t} = u_{i,j,t} + \delta_t\left(\frac{u_{i+1,j,t} - 2u_{i,j,t} + u_{i-1,j,t}}{\delta_x^2} + \frac{u_{i,j+1,t} - 2u_{i,j,t} + u_{i,j-1,t}}{\delta_y^2}\right), \tag{7}$$

which relates the temperature at a point (i, j) at time $t + \delta_t$ to the temperature at the same point and its four neighbours at time t. This equation can be used to iteratively update the temperature at each point on the grid at each time step. This process is known as explicit time integration, and it requires that the time step δ_t is small enough to ensure numerical stability.

Since a square domain is considered, it is reasonable to use a discretisation where $\delta_x = \delta_y = \delta$. Therefore, the computations would be

$$u_{i,j,t+\delta_t} = u_{i,j,t} + \delta_t\left(\frac{u_{i+1,j,t} + u_{i,j+1,t} - 4u_{i,j,t} + u_{i-1,j,t} + u_{i,j-1,t}}{\delta^2}\right). \tag{8}$$

3.2 Image denoising

The second problem that we have chosen is image denoising as it can be solved using diffusion methods. To do this, we have taken as reference the solution proposed by Tang et al. [32] to perform image denoising by applying an anisotropic diffusion method. For the sake of simplicity, we have focussed exclusively on the anisotropic flow for brightness that they proposed. Hence, to recreate this study, any grey-scale image with the same dimensions tested in these experiments is appropriate.

3.2.1 Anisotropic diffusion for brightness

A digital image is, in its fundamental form, a representation of a 2-dimensional scene as a finite array of pixels. Each pixel corresponds to a position in the scene and is assigned a value that represents the intensity or colour of the light that was recorded at that position. Therefore, a grey-scale image has only one channel, which represents the brightness or intensity of the image at each pixel. However, colour images typically have three channels, representing the red, green, and blue (RGB) components of the image at each pixel position.

A digital image can be defined as a function $f : \mathbb{R}^2 \rightarrow \mathbb{R}^c$ and we can express it as a matrix, where every pair of coordinates $M(x, y)$ is mapped to an c-dimensional vector of real numbers, representing the colour or intensity of the image at that position.

Regarding the diffusion problem, the work of Tang et al. [32] mentioned before proposed two approaches to deal with grey-scale images or brightness. The first is the isotropic flow based on the Laplace equation:

$$\frac{\partial M}{\partial t}(x,y,t) = M_{xx}(x,y,t) + M_{yy}(x,y,t) = \Delta M(x,y,t). \tag{9}$$

The second, which is the diffusion flow that we apply in this work, is the anisotropic flow, which is included in the following equation by adding the term $1/(1 + \|\nabla M\|)$:

$$\frac{\partial M}{\partial t}(x,y,t) = \frac{\left(M_{xx}M_y^2 - 2M_xM_yM_{xy} + M_{yy}M_x^2\right)^{\frac{1}{3}}}{1 + \|\nabla M\|}. \tag{10}$$

It is worth noting that solving equation (10) involves a larger number of arithmetic calculations than equation (9), which would replicate the problem explained in Sect. 3.1. Consequently, we consider two applications with different characteristics in terms of arithmetic intensity, which should allow us to explore the singularities of the studied devices.

3.2.2 Discretisation

The spatial discretisation of images is often considered trivial since the resolution of the image is already defined by its pixel grid:

$$\delta_x = \delta_y = 1. \tag{11}$$

This means that we can easily represent each point in the image as a discrete value, and the discretisation process is reduced to determining the appropriate mathematical operations to be applied to these values.

In our case, we have chosen to use an Explicit Euler method with a fixed time step to discretise the problem. By selecting a fixed time step, we can ensure that the solution remains stable and accurate throughout the simulation.

4 Implementation details

In this section, important details about the implementation of both case studies are depicted. Section 4.1 provides information about the tool used for developing both solutions, along with technical details regarding the proposed implementation. Then, the procedure for introducing dynamic load balancing is explained in Sect. 4.2.

4.1 Heterogeneous approach: Intel OneAPI

The high-level, multi-platform programming tool used to develop this work is Intel OneAPI. It is a unified software toolkit that enables the development of applications across diverse architectures. It includes compilers, libraries, and tools for developing software that can run on CPUs, GPUs, FPGAs, and other accelerators. Furthermore, OneAPI supports multiple programming languages, such as C++, Fortran, and DPC++. The latter is based on C++17 and includes a modern, heterogeneous programming model based on the SYCL standard. Intel OneAPI also includes extensive tools for profiling, debugging, and optimising code for different architectures such as Intel Advisor [33] or Intel VTune [34].

Concerning memory management, Intel OneAPI provides two ways to deal with it: USM (Unified Shared Memory) and buffers. USM provides a pointer-based approach, with a familiar syntax to C++, to allocate memory on the host (`malloc_host`), on the device (`malloc_device`), or both (`malloc_shared`). The shared allocations are useful in the case of programs where the host and the device frequently access the data since data are accessible on both. The movements of data are hidden as they are performed by runtime mechanisms and lower-level drivers. In the case of device allocations, they occur in the device-attached memory and the movement of data is explicit. This means that the developer must use copy operations to move data between the host and the device. In addition to USM, DPC++ also provides buffers, which represent a region of memory on the device. According to the SYCL 2020 Specification [35], allocations and deallocations of buffers are the responsibility of the SYCL runtime.

The program related to this study was developed using the native tools provided by Intel OneAPI mentioned above. This is, using DPC++ mechanisms to manage memory, synchronise devices, and express parallelism. In relation to the latter, SYCL handler class' `parallel_for` [36, 37] member function has been used for the GPU kernel. It provides an interface to define and invoke a SYCL kernel to execute in parallel over a specific range of values. We use a two-parameter version, with the first parameter designating the range and the second being a lambda function indicating the code that will be executed on the device. In this study, the workload distribution is performed by rows since the input data are always a matrix or an image, which is essentially the same. Thus, in our code the range is expressed with `sycl::range(size, cols)` where the `size` variable would denote the number of rows assigned to be processed on the device in that set of iterations and the `cols` variable is just the total number of columns of the input matrix or image. Section 4.2 provides a detailed explanation of how workload is distributed among devices. Similarly, the FPGA kernel follows the same strategy. Despite knowing that a specific implementation may be needed to achieve maximum performance, this study examines whether it is actually possible to generate valid kernels for different devices without modifying any lines of code using Intel OneAPI. Finally, regarding the code executed by the host, it was parallelised using OpenMP.

4.2 Dynamic load balance

In the context of problems that can be optimised using heterogeneous schemes, one of the main priorities is to perform an appropriate workload distribution among the devices involved. The aim is to maximise the benefits derived from the use of heterogeneous computing by taking advantage of the specific capabilities of each device.

We decided to use a dynamic approach for the load balancing issue. Two reasons lay behind this decision. On the one hand, using a static workload balance strategy would require either a previous profiling phase—with the consequent overhead—or a manual hand-tuning process to find the best workload share for each pair of devices. On the other hand, it is common for one of the devices to experience occasional performance drops due to system routines or some other external factors, which would deviate from an optimal share.

Therefore, since this study deals with diffusion problems that are inherently iterative, a dynamic load balancing approach has been introduced by using IHP (Iterative Heterogeneous Parallelism) [38]. This is a dynamic parallel scheme for iterative or time step methods in heterogeneous computing systems. In runtime, the workload is balanced across the devices according to their performance, aiming to improve the global execution time. The library provides a mechanism to choose which type of work distribution performs as it could be helpful to change it depending on the data. By Sect. 4.1, the work distribution in this paper is based on rows, establishing the row as the fundamental task unit.

Note that, in IHP, the parameter $\alpha \in [0, 1]$ denotes the amount of work done by the CPU. In the extrema, $\alpha = 1$ would mean that all the work is performed by the CPU and, with $\alpha = 0$ all is done by the device. For values in between, the workload is distributed among the host and the device in proportions α and $1 - \alpha$, respectively.

5 Experimental environment

All the experiments were conducted on Intel DevCloud, a cloud computing environment provided by Intel, which allows access to various CPU, GPU, and FPGA models, all manufactured by Intel. In the case of GPUs, when performing these experiments, only integrated GPUs—or iGPUs—were available in the environment. Therefore, the experiments which include a GPU were carried out exclusively on this type of device.

The specific devices used by each scheme proposed in this study can be seen in Table 1. Regarding the CPU+iGPU scheme, the specifications of the devices are: Intel Xeon E-2176 G [39] is a 6-core (12-thread) processor with a clock speed of 3.7GHz 4.7GHz and 12MB LLC (Last Level Cache); UHD Graphics P630 [39] has 24 EUs (Execution Units) and a maximum dynamic frequency of 1.2GHz. Concerning the CPU+FPGA scheme, the specifications are: Intel Xeon Gold 6128 [40] is a 6-core (12-thread) processor with a clock speed of 3.4GHz 3.7GHz and 19.25MB LLC; FPGA Arria 10 [41] has up to 1.15 million logic elements, can support up to 1500 I/O pins, supports up to 1.6TeraFLOPS of floating-point performance, and supports high-speed serial interfaces such as PCIe and 10/25/40/100 Gigabit Ethernet.

Table 1 Device models used in the experiments

Scheme	Host	Device
CPU+iGPU	Intel Xeon E-2176 G	Intel UHD Graphics P630
CPU+FPGA	Intel Xeon Gold 6128	Intel Arria 10

Intending to ensure the reproducibility of the results, the values chosen for the main program variables are detailed. This includes the matrix sizes tested, the initial CPU ratio—referred to as α in this work—and the type of memory used, among others. To begin with, the experiments were performed using square matrices ($N \times N$), where N is: 512, 800, 1024, 1500, 2048, 3000, 4096, 6000, 8192, 10000, 16384, and 20000. In the case of FPGA experiments, the largest size tested was 10000 as the execution times were higher, and that maximum size was enough to study the performance of the proposed solutions. These values were selected to span from small to large matrices, encompassing both sizes that are powers of two and those that are not. This allows us to study not only the scalability but also assess if sizes of the form 2^n are treated differently by the software or the hardware, as is often the case, implying potential performance issues. To continue, α values tested were: 0, where the device fully performs the work; 0.5, where the work is shared between the CPU and the device using dynamic workload balancing; and 1, where the CPU does all the work. The initial value of $\alpha = 0.5$ used in the mixed case was chosen assuming the ideal situation, where both devices have the same capabilities and equitable work distribution. This fact does not usually correspond to real scenarios. However, the value was chosen as a compromise solution, presuming that the dynamic load balancing strategies will correct this value during the iterations. Regarding the memory management selected, device allocations explained in more detail in Sect. 4, are used to provide more control of the data movements and the memory management. To conclude with the explanation of the parameters, all the experiments were performed using 1000 iterations to provide more accurate and reliable results and testing both cases of study: heat diffusion and image denoising.

Finally, since the host code is parallelised with OpenMP, it is important to highlight that all experiments were performed using the default value of `OMP_NUM_THREADS`, which is the maximum number of threads available for each CPU.

6 Results and discussion

This section introduces and discusses the results obtained in the experiments described in Sect. 5. Additionally, a final comparison of the execution times is presented for all the devices used in the study.

6.1 CPU+iGPU approach

This subsection includes the results obtained with the CPU Intel Xeon E-2176 G [39] and its iGPU—Intel UHD Graphics P630—for the two case studies considered.

6.1.1 Heat diffusion

Regarding the heat diffusion problem, the results obtained reveal how the CPU parallelised with OpenMP provided significantly better execution times than the iGPU for almost all the matrix sizes tested in these experiments. This is

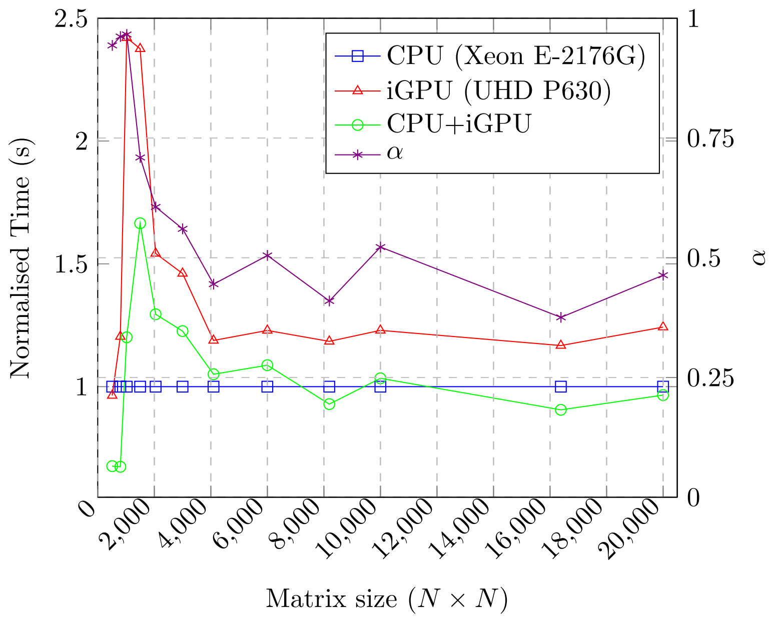

Fig. 2 CPU+iGPU: normalised times for heat diffusion. For the Y-axis on the left, lower is better

illustrated in Fig. 2, where the executing times in seconds were normalised with respect to the CPU times. When both devices are used together, in some cases the combination provides better timings in relation to the faster device. However, in most cases, the CPU-only execution is better than the hybrid scheme, which is not the expected behaviour. Concerning α, the results are coherent with the times obtained with the CPU+iGPU scheme. The initial values provide the major part of the computation to the CPU, which is to be expected considering the execution times obtained with the iGPU for these matrix sizes. But then, α is stabilised at values close to 0.5, providing a roughly equal load on both devices. This can only be translated into both devices offering similar performance when working together.

To understand the observed results, extra experiments, including the study of the time spent by each device to compute a row and the evolution of α during the iterations, were performed. For this purpose, the size $N = 6000$ was chosen since, at this value, it was expected that the GPU would improve its results, taking into account the tendency shown in Fig. 2. The results of these experiments are shown in Fig. 3. Beginning with the explanation of α evolution, as expected, the initial values provide the major part of the computation to the CPU. The distribution is logical considering the execution times obtained with the iGPU in the first approximately 10 iterations. Then, α is stabilised at values close to 0.5, providing a roughly even load on both devices. This is also a predictable behaviour since both of them seem to offer similar performance during those iterations. Considering the results obtained from running the experiments on the CPU and iGPU separately, it can be seen that the CPU provides slightly better results than the iGPU. However, when working together, the

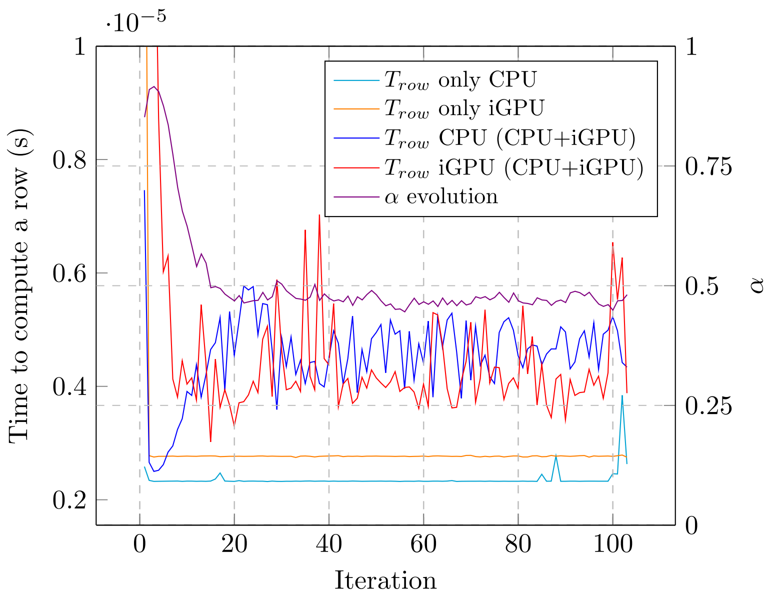

Fig. 3 T_{row} and α evolution in heat diffusion problem using only CPU, only iGPU, and CPU+iGPU for $N = 6000$

performance of both devices is degraded until they offer similar outcomes as predicted previously.

To definitively clarify this behaviour, experiments were conducted where the algebraic computational load of the problem was increased by several orders of magnitude while maintaining the same memory accesses. It was observed that in these scenarios, the iGPU improved its performance compared to the CPU by effectively exploiting its architecture. Additionally, the overhead caused by the competition between the CPU and the iGPU for memory access was reduced. In consequence, the obtained results for the mixed case improved the execution times of the fastest device. In short, the results commented on at the beginning can be attributed to the lack of enough computational load to compensate for the iGPU usage. This caused a high overhead due to the competition between the host and the device for memory access, which in this case is shared as the device is integrated.

6.1.2 Image denoising

As was explained in Sect. 3, image denoising is a more computationally demanding problem than heat diffusion, which allows the iGPU to exploit all the advantages of its architecture. Despite being an integrated device, in the context of this problem, the iGPU provides execution times significantly better than the ones obtained by the CPU for all the matrix sizes tested. Furthermore, as is expected in a theoretical scenario, the combination of both devices working together provides results that surpass the fastest device. About α, in this case, the value tends to be lower than 0.5, which means that the dynamic balancing

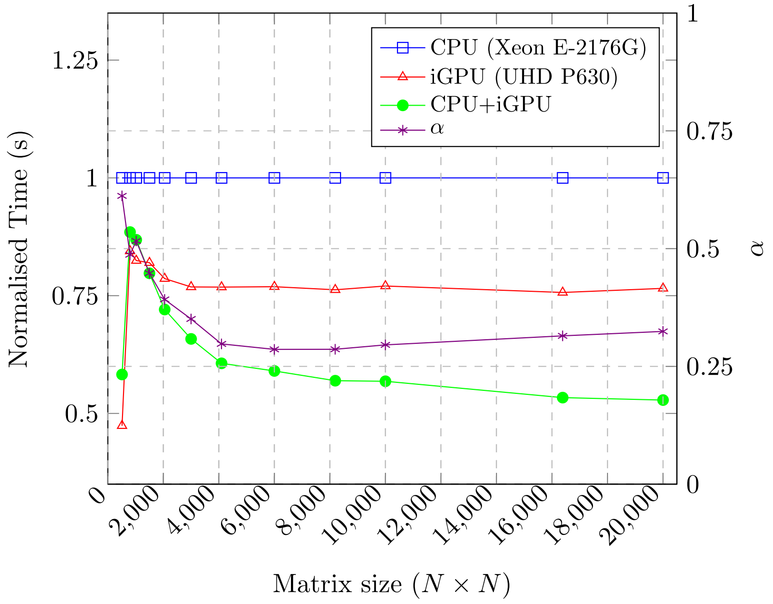

Fig. 4 CPU+iGPU: normalised execution times for image denoising

method is giving more workload to the iGPU. This work distribution is coherent since the iGPU offers better execution times than the CPU. This fact is noticeable in Fig. 4.

Therefore, this case study demonstrates that the use of heterogeneous schemes provides clear performance advantages in problems where the architecture of each device can be appropriately exploited. In contrast to the heat diffusion, Fig. 5—generated with the same parameters as the one associated with the previous case studied—allows us to observe that the iGPU's behaviour is almost unchanged when processing rows over the iterations when working alone and when working in collaboration with the CPU. Additionally, the CPU performance is actually decreased in the mixed case as it has to spend some of its time handling communications with the iGPU. In any case, the good performance of the device offsets this increase in host processing times. Concerning α, it remains pretty constant, giving most of the work to the GPU. The reason for this is that the time it takes the iGPU to process rows across iterations is also reasonably constant, allowing dynamic load balancing to be carried out quickly and with almost no change among iterations.

6.2 CPU+FPGA approach

This subsection includes the results obtained with the CPU Intel Xeon Gold 6128 [40] and the FPGA Arria 10 [41] for the two case studies considered.

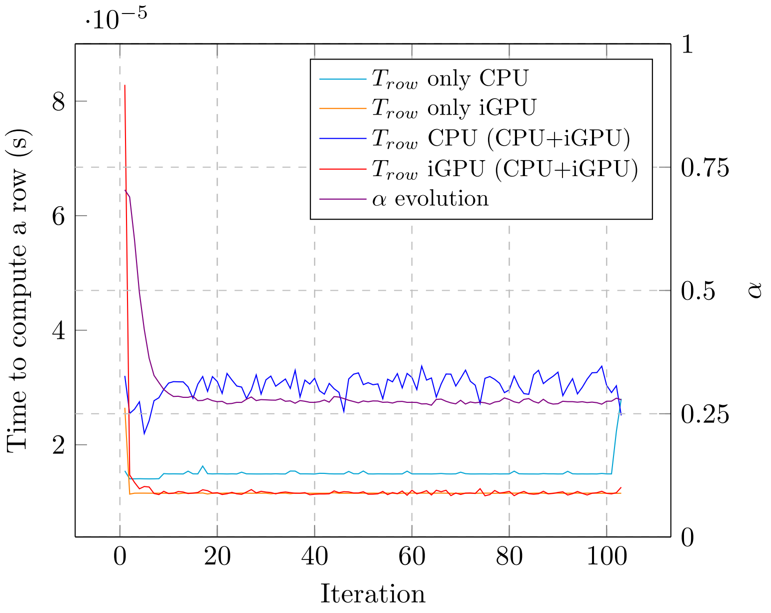

Fig. 5 T_{row} and α evolution in image denoising problem using only CPU, only iGPU, and CPU+iGPU

6.2.1 Heat diffusion

Regarding the FPGA implementation for the heat diffusion problem, it has been demonstrated that using a non-specific approach to develop the code for this type of device translates into non-competitive performance. For all the cases studied, the CPU offers significantly better execution times than the FPGA. However, it should be noted that the CPU in the FPGA node is different than the one in the GPU node, as mentioned in Sect. 5, slightly accentuating the performance differences between the devices. For the combined case where the CPU and FPGA work together, no improvement has been achieved over the results of the CPU working alone for any of the matrix sizes explored. As for α, it fluctuates consistently based on the performance of each device. The FPGA clearly shows better performance with specific matrix sizes, and it is in these cases that the library adjusts the value of α to allocate some load to the device. Yet, for sizes where the distribution of work is not advantageous, the α value assigns all the work to the CPU. Communication between the host and the device is particularly important in this case. It is noteworthy that, in this work, we use a generic kernel which can be executed in all the objective devices—avoiding device-specific code—which might result in a sub-optimal design where these transfers are very costly. Based on the results, we conclude that the high-level synthesis makes some transfers less costly than others. This behaviour is illustrated by the saw-tooth line in Fig. 6. This phenomenon is related to the fact that in the case of FPGAs, the word size used in the design has substantial implications for data transfer. This size affects the amount of data that can be sent without performing any extra padding or alignment operations, which add overhead. From the information

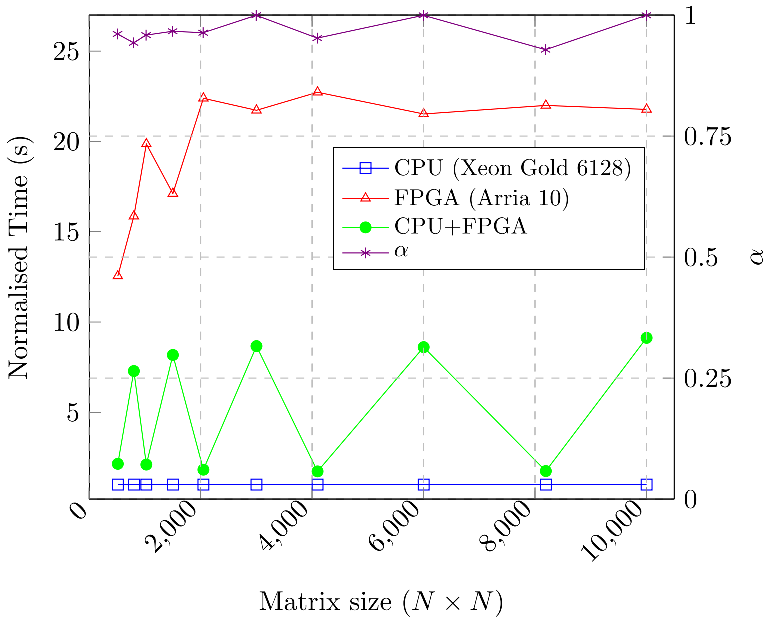

Fig. 6 CPU+FPGA: normalised execution times for heat diffusion

that could be extracted from the FPGA reports provided by the OneAPI tool—as some low-level design details are treated as a black box towards the programmer for simplicity—the data width of the DDR (Double Data Rate) memory is 512 bits. In addition to the data alignment issues, the performance drops in case of not using powers of two since the compiler always increases the size of the memory ports by using the nearest bigger power of 2 and masking out the extra bytes [42]. This fact degrades performance by wasting bandwidth. In summary, sizes that are powers of 2 provide much more competitive execution times because the time lost in host-device communications over the iterations is significantly lower than that spent with the other matrix sizes.

6.2.2 Image denoising

The results obtained with the FPGA in relation to the image denoising problem are similar to the previous ones since the CPU also provides better execution times. Nevertheless, as illustrated in Fig. 7, two notable changes affect the performance of the program. Firstly, the gap between the CPU and the FPGA is significantly less pronounced than the one observed in the context of the heat diffusion problem. Secondly, the performance of the FPGA is relatively lower than that of the CPU, but it provides more consistent results when varying the sizes of the input matrix in relation to the previous study case. This fact makes performing the dynamic workload distribution between the CPU and the FPGA easier. These two reasons explain the possibility of enhancing the results of the devices individually by utilising the

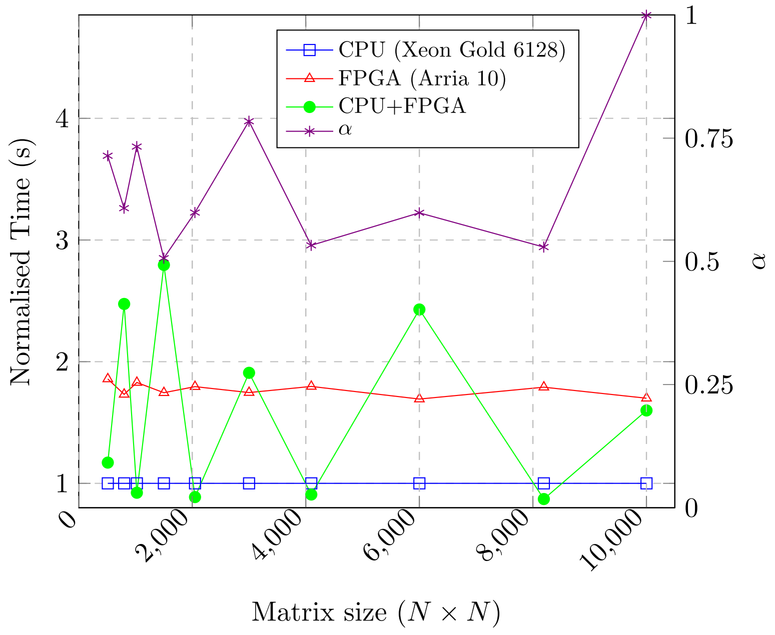

Fig. 7 CPU+FPGA: normalised execution times for image denoising

heterogeneous scheme. However, this is only feasible in experiments where the matrix size is a power of two, as commented before. In other instances, the time required for memory transfers becomes excessively large; therefore, the communication cost does not compensate for workload balancing between the host and the device.

6.3 Comparison

This section presents a set of graphical resources that facilitate the comparison of execution times among the different devices and mixed schemes explored in this study.

As depicted in Fig. 8a, in the case of the heat diffusion problem, the results obtained with CPU Xeon Gold using OpenMP with the maximum number of threads available—12 threads—were the best for cases where $N \geq 2000$. The next device to provide the best execution times is the alternative CPU model—Xeon E-2176—with 12 threads, closely followed by the CPU+iGPU scheme and the UHD P630 iGPU. Although the two CPUs evaluated have the same number of threads, as mentioned in Sect. 5, the Xeon Gold has a larger LLC size. In the case of this problem, the cache size plays an important role in speeding up data access, so the Xeon Gold performance benefits from this. Finally, the FPGA implementations using a non-specific kernel for this type of device yield the worst results. In any case, the CPU+FPGA scheme proves to be better than the FPGA working alone for all sizes.

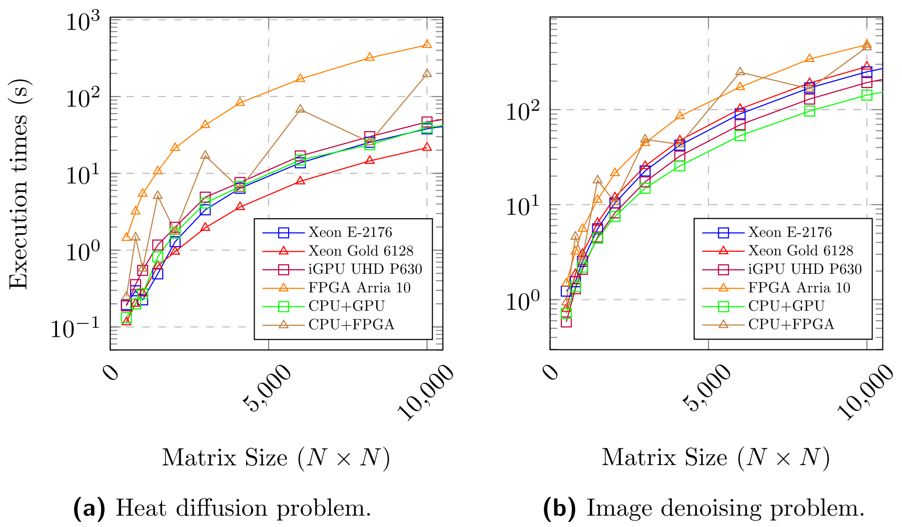

Fig. 8 Comparison of execution times for all the schemes used in the study. Lower is better

Regarding the image denoising problem, the heterogeneous CPU+iGPU scheme provided the lowest execution times, as shown in Fig. 8b. This scheme is followed by the execution of the iGPU UHD P630 and then by the two CPU models. This problem involves more arithmetic operations, thus favouring the CPU with a higher operating frequency, the Xeon E-2176 G. Once again, the worst results are obtained from the non-specific FPGA implementations. However, in this case, the CPU+FPGA scheme is not always better than the isolated execution of the Arria 10. For cases where the input matrix size is not a power of 2, it is notable that the communications between the host and the device penalise the performance significantly.

7 Conclusions

This work presents a performance-oriented study of a heterogeneous application which solves two well-known diffusion problems: heat diffusion and image denoising. With this aim, two heterogeneous schemes have been explored: CPU+iGPU and CPU+FPGA, using dynamic workload balancing strategies to improve the final performance and resource utilisation. This way, the workload is distributed between the CPU and the device in runtime. The program has been developed using the high-level tool Intel OneAPI, and the experiments were conducted on Intel DevCloud.

The results confirm the benefits of using a heterogeneous approach to solve problems which are susceptible to being parallelised following this paradigm. On the other hand, it was possible to identify the potential problems in scenarios where the architecture of any of the devices is not appropriately exploited. Moreover, all the experiments demonstrated the importance of dynamic load balancing in heterogeneous schemes due to possible performance irregularities suffered by one of the devices implicated. In any case, for the image denoising problem, we improved the execution times obtained with the fastest device—in this case, the iGPU—in $\approx 30\%$ using the CPU+iGPU scheme. Regarding the FPGA experiments, despite not developing a specific kernel for this device, the dynamic load balancing reduces the execution times of the isolated FPGA execution significantly. The best advantages obtained were 92.39% and 86.75% for heat diffusion and image denoising, respectively, in both cases where $N = 4096$. It is important to note that this advantage was only possible in cases where the memory movements between the host and the device did not imply a burden.

Regarding Intel OneAPI, it provides an option to develop heterogeneous solutions for CPUs, GPUs, and FPGAs using a unique and high-level language, DPC++. The emergence of high-level tools such as this one makes it easier to program cross-device code, facilitating and democratising access to more niche devices such as FPGAs. Nevertheless, in our experience, developing specific code for each platform considering their architectures and features is necessary to achieve optimal performance.

In future work, we plan to develop a specific FPGA solution to leverage programmable logic's advantages in these devices fully. This will enable fairer performance comparisons with the GPU-based scheme for the two study cases explored.

Acknowledgements This work has received financial support from the Consellería de Cultura, Educación e Ordenación Universitaria (accreditation ED431C 2022/16 and accreditation ED431G-2019/04) and the European Regional Development Fund (ERDF), which acknowledges the CiTIUS-Research Center in Intelligent Technologies of the University of Santiago de Compostela as a Research Center of the Galician University System. This work was also supported by the Ministry of Economy and Competitiveness, Government of Spain (Grants Nos. 28700 PID2019-104834GB-I00 and PID2022-141623NB-I00).

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Data Availability Statement Specific data are not required to reproduce this study. The work itself explains which input data are used and how to generate it.

Declarations

Conflict of interest The authors have no conflicts of interest that may have affected the content of this work.

Ethics approval Not applicable.

References

1. Nickolls J (2007) GPU parallel computing architecture and CUDA programming model. In: Proceedings of IEEE Hot chips 19 symposium (HCS), pp 1–12 https://doi.org/10.1109/HOTCHIPS.2007.7482491
2. Stone JE, Gohara D, Shi G (2010) OpenCL: a parallel programming standard for heterogeneous computing systems. Computi Sci Eng 12(3):66–73. https://doi.org/10.1109/MCSE.2010.69
3. Betkaoui B, Thomas DB, Luk W (2010) Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing. In: International Conference on Field-Programmable Technology, Beijing, China, pp 94–101. https://doi.org/10.1109/FPT.2010.5681761
4. Cong J, Fang Z, Lo M, Wang H, Xu J, Zhang S (2018) Understanding performance differences of FPGAs and GPUs. In: IEEE 26th annual international symposium on field-programmable custom computing machines (FCCM), Boulder, pp 93–96. https://doi.org/10.1109/FCCM.2018.00023
5. Che S, Boyer M, Meng J, Tarjan D, Sheaffer JW, Lee SH, Skadron K (2009) Rodinia: a benchmark suite for heterogeneous computing. In: Proceedings of the IEEE international symposium on workload characterization (IISWC), pp 44–54 .https://doi.org/10.1109/IISWC.2009.5306797
6. Vivado High-Level Synthesis (2024) https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. Accessed 12 Jan
7. Koch D, Hannig F, Ziener D (eds) (2016) FPGAs for Software Programmers. Springer. https://doi.org/10.1007/978-3-319-26408-0
8. Intel OneAPI. https://software.intel.com/content/www/us/en/develop/tools/oneapi.html. Accessed 12 Jan 2024
9. Reinders J, Ashbaugh B, Brodman J, Kinsner M, Pennycook J, Tian X (2021) Data parallel C++: mastering DPC++ for programming of heterogeneous systems using C++ and SYCL. Apress Berkeley. https://doi.org/10.1007/978-1-4842-5574-2
10. SYCL: Khronos Open Standard for C++ heterogeneous parallel programming. https://www.khronos.org/api/sycl. Accessed 12 Jan 2024
11. Lukarski D, Neytcheva M (2014) On the impact of the heterogeneous multicore and many-core platforms on iterative solution methods and preconditioning techniques. Wiley, pp 11–32. Chap. 2. https://doi.org/10.1002/9781118711897.ch2
12. Venkatasubramanian S, Vuduc RW (2009) Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In: Proceedings of the 23rd International Conference on Supercomputing (ICS). Association for Computing Machinery, New York, pp 244–255. https://doi.org/10.1145/1542275.1542312
13. Benner P, Ezzatti P, Quintana-Orti ES, Remon A (2009) Using hybrid CPU-GPU platforms to accelerate the computation of the matrix sign function. Euro-Par – Parallel Processing Workshops, pp. 132–139. Springer, Berlin, Heidelberg . https://doi.org/10.1007/978-3-642-14122-5_17
14. Benner P, Ezzatti P, Kressner D, Quintana-Ortí ES, Remón A (2011) A mixed-precision algorithm for the solution of Lyapunov equations on hybrid CPU-GPU platforms. Parallel Comput 37(8):439–450. https://doi.org/10.1016/j.parco.2010.12.002
15. Agulleiro JI, Vázquez F, Garzón EM, Fernández JJ (2012) Dynamic load scheduling on CPU-GPU for iterative tomographic reconstruction. In: IEEE 10th international symposium on parallel and distributed processing with applications, pp 603–608. https://doi.org/10.1109/ISPA.2012.90
16. Halbiniak K, Szustak L, Olas T, Wyrzykowski R, Gepner P (2021) Exploration of OpenCL heterogeneous programming for porting solidification modeling to CPU-GPU platforms. Concurr Comput Pract Exp 33(4):6011. https://doi.org/10.1002/cpe.6011
17. Belhaous S, Chokri S, Baroud S, Mestari M (2021) Comparative study of the execution time of parallel heat equation on CPU and GPU. J Commun Softw Syst 17(4):350–357. https://doi.org/10.24138/jcomss-2021-0133
18. Sánchez MG, Vidal V, Bataller J (2012) Peer group and fuzzy metric to remove noise in images using heterogeneous computing. In: Euro-Par 2011: parallel processing workshops. Springer, Berlin, Heidelberg, pp 502–510. https://doi.org/10.1007/978-3-642-29737-3_55
19. Dagum L, Menon R (1998) OpenMP: an industry standard API for shared-memory programming. IEEE Comput Sci Eng 5(1):46–55. https://doi.org/10.1109/99.660313
20. Sarjanoja S, Boutellier J, Hannuksela J (2015) BM3D image denoising using heterogeneous computing platforms. In: 2015 Conference on Design and Architectures for Signal and Image Processing (DASIP), pp. 1–8. https://doi.org/10.1109/DASIP.2015.7367257

21. Constantinescu D, Navarro A, Corbera F, Fernández-Madrigal J-A, Asenjo R (2021) Efficiency and productivity for decision making on low-power heterogeneous CPU+GPU SoCs. J Supercomput. https://doi.org/10.1007/s11227-020-03257-3
22. Yong W, Yongfa Z, Scott W, Wang Y, Qing X, Chen W (2021) Developing medical ultrasound imaging application across GPU, FPGA, and CPU using OneAPI. In: IWOCL'21. Association for Computing Machinery, New York. https://doi.org/10.1145/3456669.3456680
23. Lupescu G, Ţăpuş N (2021) Design of hashtable for heterogeneous architectures. In: 2021 23rd International Conference on Control Systems and Computer Science (CSCS), pp 172–177. https://doi.org/10.1109/CSCS52396.2021.00035
24. Marinelli E, Appuswamy R (2021) XJoin: portable, parallel hash join across diverse XPU architectures with oneAPI. In: ACM (ed) DAMON 2021, 17th International Workshop on Data Management on New Hardware, Held with ACM SIGMOD/PODS, 21 June 2021, China (Virtual Event). https://doi.org/10.1145/3465998.3466012
25. Marinelli E, Appuswamy R (2021) OneJoin: Cross-architecture, scalable edit similarity join for DNA data storage using oneAPI. In: ACM (ed) ADMS 2021, 12th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, in Conjunction with VLDB 2021, 16 August 2021, Copenhagen, Denmark, Copenhagen
26. Nozal R, Bosque JL (2021) Straightforward Heterogeneous Computing with the oneAPI Coexecutor Runtime. Electronics 10(19). https://doi.org/10.3390/electronics10192386
27. Bavarsad NN, Makrani HM, Sayadi H, Landis L, Rafatirad S, Homayoun H (2021). HosNa: a DPC++ benchmark suite for heterogeneous architectures. In: 2021 IEEE 39th International Conference on Computer Design (ICCD), pp 509–516. https://doi.org/10.1109/ICCD53106.2021.00084
28. Kashino R, Kobayashi R, Fujita N, Boku, T (2022). Multi-hetero acceleration by GPU and FPGA for astrophysics simulation on OneAPI environment. In: International Conference on High Performance Computing in Asia-Pacific Region. HPCAsia2022. Association for Computing Machinery, New York, pp 84–93. https://doi.org/10.1145/3492805.3492817
29. Groth T, Groppe S, Pionteck T, Valdiek F, Koppehel M (2023) Hybrid CPU/GPU/APU accelerated query, insert, update and erase operations in hash tables with string keys. Knowl Inf Syst 65:1–19. https://doi.org/10.1007/s10115-023-01891-w
30. Li S, Zhu J, Han J, Peng Y, Wang Z, Gong X, Wang G, Zhang J, Wang X (2023) OneGraph: a cross-architecture framework for large-scale graph computing on GPUs based on oneAPI. CCF Trans High Perform Comput. https://doi.org/10.1007/s42514-023-00172-w
31. LeVeque R (2007) Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems (classics in applied mathematics classics in applied mathematics). Society for Industrial and Applied Mathematics, New York
32. Tang B, Sapiro G, Caselles V (2000) Diffusion of general data on non-flat manifolds via harmonic maps theory: the direction diffusion case. Int J Comput Vis 36(2):149–161. https://doi.org/10.1023/A:1008152115986
33. Intel Advisor. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html. Accessed 12 Jan 2024
34. Intel V-Tune. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html. Accessed 12 Jan 2024
35. SYCL 2020 Specification. https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html. Accessed 12 Jan 2024
36. Kronos Group 1.2.1 Specification. https://registry.khronos.org/SYCL/specs/sycl-1.2.1.pdf. Accessed 12 Jan 2024
37. Intel oneAPI GPU Optimization Guide. https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2023-1/overview.html. Accessed 8 Jan 2024
38. Laso R, Cabaleiro JC, Rivera F, Muñiz FMC, Alvarez-Dios J (2021) IHP: a dynamic heterogeneous parallel scheme for iterative or time-step methods-image denoising as case study. J Supercomput 77. https://doi.org/10.1007/s11227-020-03260-8
39. Intel Xeon E-2176G Processor. https://ark.intel.com/content/www/xl/es/ark/products/134860/intel-xeon-e2176g-processor-12m-cache-up-to-4-70-ghz.html. Accessed 12 Jan 2024
40. Intel Xeon Gold 6128 Processor. https://ark.intel.com/content/www/us/en/ark/products/120482/intel-xeon-gold-6128-processor-19-25m-cache-3-40-ghz.html. Accessed 12 Jan 2024
41. Intel FPGA Arria 10. https://www.intel.la/content/www/xl/es/products/details/fpga/arria/10.html. Accessed 12 Jan 2024

42. Zohouri HR, Matsuoka S (2019) The memory controller wall: Benchmarking the Intel FPGA SDK for OpenCL memory interface. In: IEEE/ACM international workshop on heterogeneous high-performance reconfigurable computing (H2RC), pp 11–18 https://doi.org/10.1109/H2RC49586.2019.00007