
INTERNATIONAL DOCTORAL SCHOOL OF THE USC

Silvia
Rodríguez Alcaraz

PhD Thesis

# Portable heterogeneous computing on distributed HPC systems

Santiago de Compostela, 2025

**Doctoral Programme in Information Technology Research**


Ph.D. Thesis

# PORTABLE HETEROGENEOUS COMPUTING ON DISTRIBUTED HPC SYSTEMS

Silvia Rodríguez Alcaraz

Directors: Prof. Dr. David López Vilariño and Prof. Dr. Francisco Fernández Rivera

Tutor: Prof. Dr. David López Vilariño

**DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY RESEARCH**

SANTIAGO DE COMPOSTELA, 2025


# Declaration of conflicts of interest

*The doctoral candidate, Silvia Rodríguez Alcaraz, declares no conflicts of interest related to her thesis "Portable heterogeneous computing on distributed HPC systems".*

*Santiago de Compostela, 2025*

*Somentes*
*intentaba conseguir*
*deixar na terra*
*algo de min que me sobrevivise*
— Lois Pereiro, *Acróstico (1994)*

**Para Luz.**

## Agradecementos

Dende que decidín comezar esta tese, o meu día a día encheuse de retos, novas experiencias e incertezas, pero, sobre todo, dunha enorme aprendizaxe. Tiven a gran sorte de non percorrer este camiño soa, senón rodeada de xente que contribuíu, dunha forma ou doutra, a que este traballo saíse adiante.

*

Gustaríame estender un agradecemento aos meus directores: a Fran, pola súa constante implicación e valiosos consellos; e a David, por introducirme a Intel oneAPI e ao mundo das FPGAs, sendo este o xerme que propiciou todo isto. Tamén a Caba e a Tomás, grazas aos catro por compartir o voso tempo, coñecemento e asesoramento durante estes anos. Agradecer tamén aos meus compañeiros de grupo, en especial a Ruben e Miguel. Grazas Ruben, por toda a túa axuda, e polas charlas cheas de ideas e borranchos en papeis que se converten en proxectos. Espero que poidas sentir como teu un cachiño deste traballo. Grazas Miguel, por ser tan bo compañeiro dende os inicios, polas conversas sobre calquera tema e, sobre todo, por ser o mellor patiño de goma que se pode ter. Por outra banda, grazas ás compañeiras e compañeiros do CiTIUS cos que tiven a sorte de coincidir, tamén á xente das unidades de xestión e sistemas, por facernos a vida un pouco máis doada.

*

Grazas ás miñas amizades, ás de toda a vida e ás que foron chegando. Neste punto véñenme á mente moitos nomes, polo que me sinto infinitamente afortunada de estar tan ben acompañada. Por seguir abrindo e pechando etapas na vosa compañía.

*

Por último, grazas (infinitas) á miña familia. Polo voso apoio, ánimos e axuda. Polos valores que me trouxeron aquí, e que me destes vós. Especialmente, aos meus pais, María e Luis. E á miña avoa, Luz: lembrámoste.

*

Estes anos foron unha carreira de fondo que non tería rematado sen vós.

Grazas.

Santiago de Compostela, setembro de 2025.

## Additional acknowledgements

To begin with, I would like to express my gratitude to the researchers who hosted me in their groups during my time abroad. Without a doubt, collaborating with international scientific groups was an enriching experience that made me grow both professionally and personally. I am especially grateful to Gustavo Alonso for his valuable conversations, guidance, and advice during my stay at ETH Zürich in Switzerland. My sincere thanks also go to the members of the Systems Group with whom I had the luck of crossing paths, for their warm welcome, insightful talks, and inspiring company. I would also like to thank Tiago Gomes for his supervision and support during my stay with the Embedded Systems Research Group (ESRG) at the University of Minho in Portugal, as well as Ricardo Roriz for allowing me to collaborate on his PhD project and for all his help.

Furthermore, the completion of this work would have been unfeasible without the support of the various organisations and institutions that funded or provided the resources required by the experiments conducted in this thesis. Specifically, this work has received financial support from the Consellería de Cultura, Educación e Ordenación Universitaria (accreditation ED431C 2022/16 and accreditation ED431G-2023/04) and the European Regional Development Fund (ERDF), which acknowledges the CiTIUS-Research Centre in Intelligent Technologies of the University of Santiago de Compostela as a Research Centre of the Galician University System. This work was also supported by the Ministry of Economy and Competitiveness, Government of Spain (PID2019-104834GB-I00 and PID2022-141623NB-I00). In addition, the research stay at the University of Minho was funded by the 2022-2023 call of the IACOBUS program supported by the European Grouping of Territorial Cooperation of the Galicia - Northern Portugal Euroregion. Regarding the computational infrastructures, this research project was made possible through the access granted by the Galician Supercomputing Centre (CESGA) to its supercomputing infrastructure. The supercomputer FinisTerrae III and its permanent data storage system have been funded by the NextGeneration EU 2021 Recovery, Transformation and Resilience Plan, ICT2021-006904, and also from the Pluriregional Operational Programme of Spain 2014-2020

of the European Regional Development Fund (ERDF), ICTS-2019-02-CESGA-3, and from the State Programme for the Promotion of Scientific and Technical Research of Excellence of the State Plan for Scientific and Technical Research and Innovation 2013-2016 State subprogramme for scientific and technical infrastructures and equipment of ERDF, CESG15-DE-3114. Moreover, part of this work was developed using the resources allocated in the now extinct Intel Developer Cloud, access granted through the Intel DevMesh initiative.

# Contents

# Abstract

This doctoral thesis addresses the study and development of portable heterogeneous solutions, evaluating them on distributed High-Performance Computing (HPC) environments, such as cloud systems, supercomputers, and clusters.

Since its beginnings, the history of computing has been shaped by milestones aimed at enhancing the capabilities of computing systems to meet the challenges of each era. These improvements have involved both hardware optimisations and the development of programming techniques and algorithms designed to make the most effective use of those platforms. In the late 1970s, the advent of computer networks enabled the creation of distributed computing environments capable of combining the power of multiple machines to create more complex and powerful systems. The final years of the 20th century were characterised by efforts to reduce the size of chip components, mainly transistors, a process known as miniaturisation. This made it possible to include more transistors per chip, thereby increasing their processing capacity, as predicted by Moore's Law, formulated in 1965. In the early 2000s, single-processor machines evolved into multicore systems. For some time, the miniaturisation of chip components, together with the fabrication of processors with a higher number of cores and threads, was the main strategy to achieve greater computational performance. These advances in hardware and infrastructure were accompanied by developments in software. Examples include the Open Multi-Processing (OpenMP) library, used to implement parallel programmes on shared-memory multicore machines, and the Message Passing Interface (MPI) paradigm, designed to distribute workloads across the computing nodes of distributed systems. Between 2005 and the early 2010s, the so-called post-Moore era began, marked by a growing shift towards heterogeneous systems in response to the increasing computational demands of modern applications. On the one hand, this change was driven by the physical limitations that slowed the progression predicted by Moore's Law, and by the end of transistor scaling as described by Dennard's rule. On the other hand, current applications not only demand higher computational power but also exhibit highly specific workloads. The inference tasks performed by deep learning models are a clear example of this. These two factors have largely determined the adoption of specialised hardware to achieve both greater computational performance and higher system specialisation, leading to the rise of

heterogeneous architectures.

Chapter 2 provides contextualisation for the two main topics related to the thesis: heterogeneous computing and distributed computing systems. Section 2.1 introduces heterogeneous computing, explaining the advantages and challenges associated with this paradigm. It also provides a detailed description of the evolution and current state of the specific hardware studied in this work: Graphics Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs), together with the development tools available for their programming. Nowadays, there is a wide range of specialised devices, often referred to as "accelerators", for creating heterogeneous applications. In this context, there is a host, usually responsible for tasks related to programme coordination. In general, the Central Processing Unit (CPU) assumes this role. Furthermore, there is a device, which is a specific accelerator to which complex computations are offloaded in order to improve system performance. The code executed on these devices is usually referred to as a kernel. Despite the diversity of available accelerators, GPUs remain the most widely used. These devices were introduced in the mid-1990s to accelerate calculations related to graphics processing. Due to the potential of their highly parallel architecture, the programmability of GPUs was enhanced to accelerate other types of tasks beyond graphics, a trend known as General-Purpose Computing on GPU (GPGPU). Moreover, the development of Compute Unified Device Architecture (CUDA) in 2006 provided the developer community with the necessary tools to program NVIDIA GPUs using high-level languages such as C++. This led to their growing popularity and widespread adoption in HPC applications. Although GPUs are the most commonly used accelerators due to their high degree of parallelism and the large ecosystem of supporting tools, other devices are also worth mentioning. With an even more specialised focus, there are Neural Processing Units (NPUs) and Tensor Processing Units (TPUs), the latter introduced by Google in 2016 to accelerate tensor-based operations. These devices play an especially relevant role in the field of Artificial Intelligence (AI), as they are primarily designed to accelerate the characteristic workloads present in such applications. NPUs and TPUs exemplify the ongoing trend towards greater hardware specialisation. In addition, it is worth highlighting the role of FPGAs, which were introduced to the market in the late 1980s. These reprogrammable devices were initially used as an alternative to Application Specific ICs (ASICs) in cases where system production was not particularly high, since in such contexts the manufacturing costs of an ASIC were prohibitive. Over time, FPGAs evolved to improve their computational capabilities, combining a Programmable Logic (PL) with a Processing System (PS), typically equipped with an Advanced RISC Machine (ARM). The main advantage of these devices lies in their reconfigurability, which allows them to be programmed specifically for a given problem. In certain cases, this results in a significant improvement in performance as well as better resource utilisation, translating into energy savings. However, a well-known challenge associated with FPGAs since their introduction has been the intrinsic difficulty of their programming. The traditional method involves using Hardware Description Languages (HDLs), re-

quiring knowledge of digital systems and low-level programming to design efficient solutions. Today, there are tools capable of translating code written in high-level languages into HDL, a process known as High-Level Synthesis (HLS). Nevertheless, an optimal design written directly in HDL generally achieves higher performance, meaning that choosing an HLS tool often entails sacrificing some degree of optimisation in exchange for greater programmability. This reflects the different levels of maturity between FPGAs and other devices in terms of high-level development tools. Another aspect related to development tools, which in this case affects heterogeneous computing in general, is the challenge posed by the vast diversity of accelerators available today. Typically, each device is programmed using its own tool or framework, often dependent on a specific vendor. This makes the maintenance and development of codes implementing heterogeneous systems more complex. Motivated by this issue, cross-platform programming models have emerged, such as SYCL, proposed by the Khronos Group. These models aim to provide a unified abstraction that allows developers to create applications for multiple platforms using a single source code. Apparently, this approach solves the problem; however, a fundamental aspect is the evaluation of the performance offered by such solutions compared with proprietary and device-specific tools such as CUDA. For this purpose, there exists a metric known as performance portability, which provides a qualitative way to assess the portability of the performance of an application implemented with a given tool across a set of hardware platforms.

The trend towards heterogeneity affects both personal computers and large-scale systems. The second part of Chapter 2, found in Section 2.2, examines distributed computing systems, with an emphasis on HPC-oriented environments, and analyses the growing presence of heterogeneous hardware in such contexts. A distributed system can be defined as a collection of independent machines that, from the user's perspective, operate as a single system. The individual computing units are called nodes, which are interconnected through a communication network. In addition, the software layer responsible for abstracting the system's complexity and serving as the user interface is known as the middleware. These systems also exhibit a set of properties that distinguish them from domestic device networks, among which accessibility, abstraction, security, and scalability stand out. The approach adopted in the design and development of these systems is also fundamental, with several paradigms of distributed computing existing. The main ones in the context of HPC are computing clusters, grid systems, and cloud computing systems. In this work, a cloud system, Intel Developer Cloud, and a supercomputer, the Finisterrae III (FT3) maintained by the Centro de Supercomputación de Galicia (CESGA), were used to conduct the experiments related to the performance analysis of the developed solutions. Starting with clusters, their popularity is linked to the advances achieved during the 1980s and 1990s in microprocessor performance, interconnection network speed, the standardisation of distributed programming tools, and the high computational demand of contemporary scientific applications. Formally, a cluster is a collection of interconnected

independent machines that work collaboratively as a single computational resource. A supercomputer can be considered a type of cluster designed to achieve the highest possible computational power, providing scientific organisations and institutions with the necessary resources to solve complex problems. Both clusters and supercomputers were initially homogeneous, composed of CPU-based nodes, but their evolution towards heterogeneity is now evident. An analysis of the Top500 list, which compiles data on the world's most powerful supercomputers, shows that in 2015 only 20% of these systems were heterogeneous. Ten years later, this percentage has risen to 47%. Furthermore, NVIDIA GPUs are present as accelerators in 85% of heterogeneous systems, highlighting the dominance of both these devices and their manufacturer within the field of HPC. Today, the aggregate computational power of heterogeneous systems surpasses that of homogeneous ones, as does their energy efficiency. This demonstrates the advantages that dedicated devices can offer, both in terms of computational performance and in reducing carbon footprint. Regarding clouds, these are distributed systems that provide ubiquitous, scalable, and on-demand access to a pool of computing resources. To meet the needs of different use cases, three main service models exist: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Additionally, depending on their organisation, cloud systems may be public, private, community-based, or hybrid, the latter combining several deployment models. In terms of heterogeneity, it can be said that nearly all, or at least the most popular clouds, provide access to both CPU and GPU devices. This is the case for Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), Alibaba Cloud, and Intel Tiber AI Cloud, most of which also offer access to devices from multiple vendors as well as AI accelerators. Moreover, AWS, Azure, and Alibaba allow the reservation of instances equipped with FPGAs, devices that, although not yet widely used in mainstream software development, are gaining recognition due to the benefits offered by their reconfigurability. Specifically, FPGAs can process data directly on the network, without requiring transfers or communication with the CPU, which is highly advantageous in such systems. The integration of diverse hardware and the emergence of new application domains with novel computational challenges, including the Internet of Things (IoT), have led in recent years to the rise of new paradigms such as edge computing and fog computing. Edge computing aims to bring processing closer to the data source in order to accelerate computation, which can be critical in applications with real-time constraints. Fog computing, in turn, combines this approach with the transfer of part of the data or results to remote servers, such as clouds or clusters, for further processing.

Given the previous context on heterogeneous computing on distributed systems, Chapter 3 presents a study in which the performance of heterogeneous CPU+GPU and CPU+FPGA configurations is evaluated across different computing environments, including a cloud and a supercomputer. To this end, a microbenchmark was developed using Intel oneAPI, Intel's implementation of SYCL, with the aim of creating a portable solution. The case studies considered two iterative problems: two-

dimensional heat diffusion and image denoising using an anisotropic method. Specifically, the first represents a memory-bound problem, while the second is computationally intensive. A dynamic load-balancing technique was also applied between the host (CPU) and the device (the GPU or FPGA, depending on the case). For this purpose, the Iterative Heterogeneous Parallelism (IHP) library was employed, in which the parameter $\alpha \in [0, 1]$ determines, at runtime, the workload assigned to each part based on the performance achieved during a previous set of iterations. A value of 0 indicates that all work is assigned to the device, while 1 means that all work is carried out by the CPU. Intermediate values define how the workload is distributed between the two, with $1 - \alpha$ for the device and $\alpha$ for the CPU. In this study, a row-based partitioning strategy was used with an initial value of $\alpha = 0.5$. Performance tests were conducted on four different configurations, where the accelerators used were: an Integrated GPU (iGPU), an NVIDIA Discrete GPU (dGPU) designed for HPC, a high-end NVIDIA dGPU for desktop systems, and an Intel FPGA. Support for NVIDIA GPUs, not natively available in Intel oneAPI, was enabled through integration of Codeplay's plugin for NVIDIA devices. The experiments were carried out using $N \times N$ matrices with values of $N$ ranging from 512 to 20 000, including powers of two as well as intermediate values to investigate the performance implications of problems where $N = 2^n$. Data were represented using 32-bit floating-point precision. Memory was managed using Intel oneAPI's Unified Shared Memory (USM) mechanisms, and parallelism within kernels was expressed through the *parallel_for* clause. Additionally, the host code was parallelised using the OpenMP library, employing the maximum number of cores available on each machine. Regarding the results, for the memory-bound problem it was observed that the dGPUs achieved the highest performance, primarily due to their larger and dedicated device memory. Furthermore, with such powerful accelerators, the heterogeneous scheme did not outperform the use of the dGPU alone, as the overhead associated with workload distribution was not offset. Conversely, the iGPU exhibited limited performance, which can be attributed to its shared memory with the CPU. For the computationally intensive problem, the dGPUs again demonstrated high performance, but in this case, the heterogeneous configuration achieved better execution times than the accelerator alone when the CPU's contribution compensated for the time spent on load rebalancing. The iGPU also outperformed the CPU in this scenario due to its parallel architecture, and the heterogeneous CPU+GPU configuration provided an advantage over executing the fastest device independently. Finally, it is worth noting that when using a common codebase for all devices to evaluate oneAPI's portability, the high-level synthesis generated from the FPGA kernels did not yield competitive results in any of the evaluated cases. An important observation is that, for these devices, the use of problem sizes following the form $2^N$ proved critical. This is because the HLS tool generates data buses in powers of two, meaning that intermediate data sizes require additional padding or alignment operations, which substantially increase transfer times.

Chapter 4 evaluates Intel's automatic CUDA-to-SYCL migration tool, known as

SYCLomatic, or DPC++ Compatibility Tool (DPCT) in its commercial version included within the oneAPI toolkit. For this purpose, the CUDA implementation of the Rodinia benchmark suite was migrated. Rodinia is a well-known collection of programs for assessing the performance of heterogeneous applications, and its latest supported version dates back to 2014. For this reason, minor modifications were made to the codes without modifying their functionality. These adjustments were necessary to enable execution with a modern version of CUDA. Standardised timing measurements were also introduced to analyse the execution times of the different parts of the codes and to allow precise comparisons. To assess the degree of automation provided by the tool, all Rodinia programs written in CUDA were automatically migrated to SYCL, achieving a 93.24% success rate. The tool also generates a set of warning messages containing information about the migration process, details that should be reviewed by the developer, and notifications concerning performance, potential incompatibilities, or other relevant issues. The obtained warnings were classified into categories according to their function, revealing that the most frequent ones were related to differences between the Application Programming Interfaces (APIs) of CUDA and oneAPI (35.24%), performance-related aspects (27.81%), and error handling (17.41%). It was also found that the number of warnings produced was not directly correlated with the number of lines of code to be migrated, indicating that the dimension of the initial program does not determine the success of the migration. The analysis suggests that other factors may be involved, probably related to the availability of suitable SYCL equivalents or the absence of unsupported routines. Regarding performance, experiments were conducted on two NVIDIA GPU nodes of the FT3 supercomputer. Each experiment consisted of 5 initial warm-up iterations, discarded and used solely for system initialisation, followed by 10 real measurements to ensure the reliability of the results. In most of the evaluated benchmarks, the CUDA and migrated SYCL implementations achieved comparable execution times on both tested GPUs. Likewise, the obtained performance portability values were suitable for GPU platforms and were slightly higher for SYCL. To evaluate this aspect, two metrics were employed: $\Phi$, which in recent years has become the standard within the HPC community for such studies; and $\overline{\Phi}$, which penalises performance imbalance between platforms less severely and has begun to be adopted in cases where performance portability is computed based on application efficiency rather than architectural efficiency, as is the case in this study.

Among the conclusions drawn from this work, the most prominent is the clear trend towards hardware heterogeneity across all scales of computing systems, together with the computational challenges that this scenario presents, including the need to standardise development tools for heterogeneous software. In this context, the study presented in Chapter 3 confirmed the feasibility of the SYCL implementation developed by Intel, oneAPI, for programming this type of application. Nevertheless, the implementation of device-specific kernels is recommended when performance, in addition to portability, is a priority. This study also revealed the different levels of

maturity among accelerators, with FPGAs still lagging behind GPUs in terms of high-level development capabilities. Enhancing the ecosystem of tools surrounding these devices will be crucial for their wider adoption and democratisation, since their inherent characteristics give them enormous potential in the future of heterogeneous HPC systems. Furthermore, the emergence of automatic migration technologies, such as the one evaluated in Chapter 4, facilitates the transition from proprietary models to cross-platform programming paradigms.

# Resumo

Nesta tese de doutoramento abórdase o estudo e desenvolvemento de solucións heteroxéneas portables, avaliándoas en contornas de computación de alto rendemento ou High-Performance Computing (HPC) distribuídas, como poden ser os sistemas de computación na nube, os supercomputadores ou os *clusters*.

Dende os seus inicios, a historia da computación estivo marcada por fitos orientados a mellorar as capacidades dos sistemas de cómputo para poder facer fronte aos retos existentes en cada época. Estas melloras implicaron tanto optimizacións do propio *hardware* como o desenvolvemento de técnicas de programación e algoritmos destinados a aproveitar da forma máis adecuada esas plataformas. A finais da década de 1970, a introdución das redes de computadores permitiu a creación de contornas de cómputo distribuídas, capaces de combinar a potencia de múltiples máquinas para formar sistemas máis complexos e potentes. Os últimos anos do século XX estiveron marcados polos esforzos orientados a reducir o tamaño dos compoñentes dos chips, principalmente dos transistores, o que se coñece como miniaturización. Este proceso permitía introducir máis transistores por chip, aumentando a súa capacidade de procesamento tal e como predicía a Lei de Moore postulada en 1965. A comezos dos anos 2000, os equipos cun único procesador evolucionaron ata se converteren en sistemas multinúcleo. Durante un tempo, a miniaturización dos compoñentes electrónicos dos chips xunto coa fabricación de procesadores cun maior número de núcleos e fíos foi a principal estratexia para acadar un maior rendemento computacional. Estes avances a nivel de *hardware* e de infraestrutura estiveron acompañados por desenvolvementos *software*. Exemplos disto son a librería Open Multi-Processing (OpenMP), para implementar programas paralelos en máquinas multinúcleo con memoria compartida, ou o paradigma Message Passing Interface (MPI), orientado a repartir as cargas de traballo entre os distintos nodos de cómputo dos sistemas distribuídos. Entre 2005 e os inicios da década de 2010, comeza a denominada época post-Moore, observándose unha crecente tendencia cara os sistemas heteroxéneos co obxectivo de dar resposta á crecente demanda computacional das aplicacións modernas. Este cambio débese, por unha parte, ás limitacións físicas que frearon a evolución prevista pola Lei de Moore, así como ao esgotamento da escalabilidade segundo a regra de Dennard. Por outra parte, as aplicacións actuais non só requiren maior potencia de cómputo, senón

que tamén presentan cargas de traballo específicas. As tarefas de inferencia realizadas polos modelos de aprendizaxe profundo supoñen un claro exemplo disto. Estes dous factores son os que principalmente ditaminan o uso de *hardware* específico para lograr tanto unha maior potencia de cómputo como unha maior especialización por parte dos sistemas, desembocando no auxe das arquitecturas heteroxéneas.

O capítulo 2 proporciona contextualización sobre os dous temas principais nos que se centra a tese: computación heteroxénea e sistemas de computación distribuídos. A sección 2.1 introduce a computación heteroxénea, explicando as vantaxes e os retos que supón este paradigma. Ademais, descríbese en detalle a evolución e actual estado do *hardware* específico estudado neste traballo, as unidades de procesamento gráfico ou Graphics Processing Units (GPUs) e as matrices de portas lóxicas reprogramables ou Field Programmable Gate Arrays (FPGAs), xunto coas ferramentas de desenvolvemento dispoñibles para a súa programación. Hoxe en día, existe un amplo abano de dispositivos especializados, tamén denominados co termo "aceleradores", para crear aplicacións heteroxéneas. Neste contexto, existe un dispositivo *host*, normalmente encargado de tarefas relacionadas coa coordinación do programa. Polo xeral, a unidade central de procesamento ou Central Processing Unit (CPU) é a que toma este rol. Por outra banda, está o *device* ou dispositivo, que é un acelerador específico ao que se derivan os cómputos complexos co fin de mellorar o rendemento do sistema. O código que se executa nos dispositivos acostuma denominarse como *kernel*. Pese á diversidade de aceleradores mencionada, as GPUs seguen sendo os máis comunmente utilizados. Estes dispositivos foron introducidos no mercado a mediados dos anos 1990 para acelerar cálculos relacionados co procesamento de gráficos. Debido ao potencial da súa arquitectura altamente paralela, a programabilidade das GPUs mellorouse para poder acelerar outro tipo de tarefas e non só gráficos, tendencia coñecida como computación de propósito xeral en GPU ou General-Purpose Computing on GPU (GPGPU). Ademais, o desenvolvemento de Compute Unified Device Architecture (CUDA) en 2006 aprovisionou á comunidade de desenvolvedores coas ferramentas necesarias para programar códigos para GPUs de NVIDIA mediante linguaxes de alto nivel como C++. Isto propiciou o crecemento da súa popularidade, adoptándose de forma masiva en aplicacións de HPC. Aínda que as GPUs son os aceleradores máis utilizados, pola súa arquitectura altamente paralela e o grande ecosistema de ferramentas que as rodea, existen outros dispositivos a ter en conta. Cun foco aínda máis específico que o das GPUs, están as unidades de procesamento neuronal ou Neural Processing Units (NPUs) e as unidades de procesamento tensorial ou Tensor Processing Units (TPUs), sendo estas últimas introducidas por Google en 2016 para acelerar operacións baseadas en tensores. Estes dispositivos cobran un papel especialmente relevante no ámbito da intelixencia artificial ou Artificial Intelligence (AI), xa que están principalmente enfocados en acelerar as cargas de traballo específicas que conteñen este tipo de aplicacións. As NPUs e as TPUs exemplifican a tendencia que existe cara a maior especialización do *hardware*. Por outra banda, convén destacar o papel das FPGAs, introducidas no mercado a finais da década de 1980. Estes dispositivos reprograma-

bles foron empregados nos seus inicios como alternativa aos circuítos integrados de aplicación específica ou Application Specific ICs (ASICs) cando a produción do sistema non era especialmente elevada, xa que neses contextos os gastos de produción dun ASIC eran inviables. Pouco a pouco, as FPGAs foron evolucionando ata mellorar as súas capacidades de cómputo, combinando unha parte de lóxica programable, ou Programmable Logic (PL), cun sistema de procesamento, ou Processing System (PS), xeralmente dotado dunha máquina con arquitectura RISC avanzada ou Advanced RISC Machine (ARM). A principal vantaxe destes dispositivos reside na súa reconfigurabilidade, o que lles permite ser programados de forma específica para un problema concreto. En certos casos isto supón unha importante mellora en canto ao rendemento, así como un maior aproveitamento dos recursos, que se traduce en aforro enerxético. Un problema ligado ás FPGAs dende os seus inicios é a dificultade intrínseca que supón a súa programación. O método tradicional é utilizando linguaxes de descrición *hardware* ou Hardware Description Languages (HDLs), sendo necesario posuír coñecementos sobre sistemas dixitais e programación a baixo nivel para deseñar solucións eficientes. Hoxe en día existen ferramentas que traducen código escrito con linguaxes de alto nivel a HDL, o que se coñece como síntese de alto nivel ou High-Level Synthesis (HLS). De todas formas, un deseño óptimo con HDL xeralmente proporciona máis rendemento, polo que a elección dunha ferramenta de HLS implica sacrificar un maior nivel de optimización por unha maior programabilidade. Isto amosa o distinto grao de madurez das FPGAs fronte a outros dispositivos en canto a ferramentas de desenvolvemento de alto nivel. Outro aspecto relacionado coas ferramentas de desenvolvemento que, neste caso, afecta á computación heteroxénea en xeral é o reto que supón a gran diversidade de aceleradores existentes hoxe en día. Normalmente, cada dispositivo é programado mediante a súa propia ferramenta ou *framework*, en moitas ocasións dependente dun vendedor específico. Isto dificulta o mantemento dos códigos que implementan sistemas heteroxéneos, así como o seu desenvolvemento. Con esta motivación xorden os modelos de programación multiplataforma, como é o caso de SYCL, proposto polo Khronos Group. Estes modelos teñen por obxectivo proporcionar unha abstracción unificada que permita desenvolver aplicacións para múltiples plataformas empregando un mesmo código fonte. Aparentemente, este enfoque resolve o problema, pero un aspecto fundamental é a avaliación do rendemento ofrecido por este tipo de solucións en relación ao de ferramentas propietarias e específicas como CUDA. Para isto existe unha métrica denominada *performance portability*, a cal proporciona unha forma cualitativa de avaliar a portabilidade do rendemento dunha aplicación implementada cunha ferramenta determinada e con respecto a un conxunto de plataformas *hardware*.

A tendencia cara a heteroxeneidade afecta tanto aos equipos persoais como aos sistemas a grande escala. A segunda parte do capítulo 2, que se atopa na sección 2.2, trata sobre os sistemas de computación distribuídos, con énfase nos sistemas orientados a HPC, e amosa de modo analítico a crecente presenza de *hardware* heteroxéneo neste tipo de contornas. Un sistema distribuído pódese definir como un conxunto de máquinas

independentes que, para o usuario, funcionan como un único sistema. Neste contexto, as unidades individuais de cómputo denomínanse nodos, os cales están conectados mediante unha rede de comunicación. Por outra banda, a capa de *software* encargada de abstraer as complexidades do sistema e coa que interactúa o usuario denomínase *middleware*. Este tipo de sistemas, ademais, posúen unha serie de características que os distingue dunha rede doméstica de dispositivos. Entre elas destacan a accesibilidade, a abstracción, a seguridade e a escalabilidade. O deseño seguido para crear estes sistemas tamén é fundamental, existindo distintos paradigmas de computación distribuída. Os principais en relación ao ámbito de HPC son os *clusters* de computación, os sistemas *grid* e os sistemas de computación na nube ou *clouds*. Neste traballo empregouse un sistema *cloud*, Intel Developer Cloud, e un supercomputador, o Finisterrae III (FT3), mantido polo Centro de Supercomputación de Galicia (CESGA), para a realización dos experimentos ligados á análise do rendemento das solucións desenvolvidas. Comezando polos *clusters*, a súa popularidade está vinculada aos avances durante as décadas de 1980 e 1990 en relación ao rendemento dos microprocesadores, á velocidade das redes de interconexión, á estandarización de ferramentas de programación distribuída e á alta demanda computacional das aplicacións científicas do momento. Formalmente, un *cluster* é un conxunto de máquinas independentes interconectadas que traballan de forma colaborativa como un único recurso computacional. Pode dicirse que un supercomputador é un tipo de *cluster* orientado a posuír a maior potencia computacional posible para dotar a organizacións e entidades do ámbito científico dos recursos necesarios para resolver problemas complexos. Tanto os *clusters* como os supercomputadores eran inicialmente homoxéneos, formados por nodos de CPU, pero a súa tendencia cara a heteroxeneidade é evidente. Analizando o Top500, unha lista que recolle os datos dos supercomputadores máis potentes do mundo, determinouse que no 2015 só o 20 % destes sistemas eran heteroxéneos. Dez anos despois, na actualidade, esta porcentaxe alcanza o 47 %. Tamén se detectou a presenza de GPUs de NVIDIA como aceleradores no 85 % dos sistemas heteroxéneos, mostrando un claro dominio tanto destes dispositivos como do vendedor no ámbito da computación de alto rendemento. Ademais, hoxe en día a potencia de cómputo agregada polos sistemas heteroxéneos supera á dos homoxéneos, ao igual que sucede coa eficiencia enerxética. Este feito pon de manifesto as vantaxes que os dispositivos dedicados poden ofrecer, tanto para obter potencia computacional como para reducir a pegada de carbono. En canto aos *clouds*, son sistemas distribuídos que proporcionan un acceso ubicuo, escalable e baixo demanda a un conxunto de recursos computacionais. Para adaptarse a distintos casos de uso, existen tres modelos de servizo principais: infraestrutura como servizo ou Infrastructure as a Service (IaaS), plataforma como servizo ou Platform as a Service (PaaS) e *software* como servizo ou Software as a Service (SaaS). Ademais, dependendo de como estea organizado, un sistema *cloud* pode ser público, privado, comunitario ou híbrido, no caso de combinar varios modelos de organización. En canto á heteroxeneidade dos *clouds*, pódese dicir que na súa totalidade, ou polo menos nos sistemas máis populares, é posible conseguir acceso a dispositivos CPU e GPU. Isto

é así para Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), Alibaba Cloud e Intel Tiber AI Cloud. A maioría proporcionando tamén acceso a dispositivos de varios vendedores, así como a aceleradores de AI. Ademais, tanto AWS como Azure e Alibaba permiten a reserva de instancias con FPGAs, dispositivos que, pese a non ter un uso tan estendido no desenvolvemento de *software mainstream*, están gañando notoriedade debido ás vantaxes que a súa reconfigurabilidade pode ofrecer. En concreto, as FPGAs son capaces de procesar datos directamente na rede, sen necesidade de transferencias ou comunicacións coa CPU, o cal é moi beneficioso neste tipo de sistemas. Debido á incorporación de *hardware* diverso e á aparición de ámbitos de aplicación con novos retos computacionais, como o sector do Internet das cousas ou Internet of Things (IoT), nos últimos anos emerxeron novos paradigmas como a computación no bordo ou *edge computing* e a computación na néboa ou *fog computing*. O primeiro paradigma busca achegar o procesamento á fonte dos datos para acelerar os cálculos, o cal pode ser determinante en aplicacións con certo tipo de restricións, como son aquelas que deben funcionar en tempo real. Pola súa parte, *fog computing* combina o anterior co envío de certa parte dos datos ou dos resultados a servidores remotos, como *clouds* ou *clusters*, para o seu procesamento.

Rematado o contexto previo sobre computación heteroxénea en sistemas distribuídos, o capítulo 3 presenta un estudo no cal se avalía o rendemento de configuracións heteroxéneas CPU+GPU e CPU+FPGA en distintas contornas de computación, incluíndo un *cloud* e un supercomputador. Para isto desenvolveuse un *microbenchmark* con Intel oneAPI, a implementación de SYCL proporcionada por Intel, co fin de crear unha solución portable. Os casos de estudo empregados foron dous problemas iterativos: a difusión 2D do calor e a eliminación de ruído en imaxes mediante un método anisótropo. En concreto, o primeiro supón un problema onde a limitación vén dada pola memoria, mentres que o segundo é un problema computacionalmente intensivo. Tamén se utilizou unha técnica de balanceo de carga dinámico entre o *host* (CPU) e o dispositivo (a GPU ou FPGA, segundo o caso). Para isto empregouse a libraría Iterative Heterogeneous Parallelism (IHP), na cal o parámetro $\alpha \in [0, 1]$ determina a carga de traballo correspondente a cada unha das partes en tempo de execución tendo en conta o rendemento acadado nun conxunto de iteracións anterior. En concreto, o valor 0 indica que todo o traballo é asignado ao dispositivo, mentres que 1 significa que todo o traballo é realizado pola CPU. Valores intermedios determinan a forma na que se reparten as tarefas entre ambas partes, da forma $1 - \alpha$ para o dispositivo e $\alpha$ para a CPU. No estudo empregouse unha repartición baseada en filas cun valor inicial de $\alpha = 0,5$. As probas de rendemento realizáronse sobre catro configuracións distintas, onde os aceleradores empregados foron: unha GPU integrada ou Integrated GPU (iGPU) de Intel, unha GPU discreta ou Discrete GPU (dGPU) de NVIDIA orientada a HPC, unha dGPU de NVIDIA de alta gama para escritorio, e unha FPGA de Intel. O soporte para GPUs de NVIDIA, non nativo en Intel oneAPI, obtívose mediante a integración do *plugin* para dispositivos de NVIDIA de Codeplay. Seguindo coas probas, utilizáronse matrices $N \times N$ explorando valores de $N$ comprendidos entre 512 e

20 000, incluíndo potencias de dous e valores intermedios para estudar as implicacións no rendemento de problemas onde $N = 2^n$. Sobre a precisión dos datos, usáronse números en punto flotante de 32 bits. Por outra banda, empregáronse os mecanismos de oneAPI baseados en memoria compartida unificada ou Unified Shared Memory (USM) para xestionar a memoria e a cláusula *parallel_for* para expresar paralelismo nos *kernels*. Ademais, o código da parte *host* paralelizouse coa librería OpenMP, empregando o máximo número de núcleos de cada máquina para os experimentos. En canto aos resultados, para o problema limitado por memoria, determinouse que as dGPUs ofrecían o maior rendemento debido á súa memoria, de maior tamaño e reservada para uso exclusivo do dispositivo. Ademais, con este tipo de aceleradores tan potentes, o esquema heteroxéneo non amosou melloras fronte ao uso exclusivo da dGPU, xa que non se chegaba a compensar o tempo empregado na repartición de traballo. Pola contra, a iGPU mostrou unha limitación de rendemento debido ao feito de compartir a memoria coa CPU. No caso dun problema computacionalmente intensivo, de novo, as dGPUs demostraron un gran rendemento, pero neste caso a configuración heteroxénea logrou mellorar a execución independente do acelerador cando o traballo aportado pola CPU compensaba o tempo invertido nos rebalanceos de carga. A iGPU tamén superou á CPU neste escenario gracias á súa arquitectura paralela e, ademais, a solución heteroxénea CPU+GPU proporcionou unha vantaxe fronte á execución independente do dispositivo máis rápido. Por último, é relevante mencionar que ao empregar un código común para todos os dispositivos co fin de avaliar a portabilidade ofrecida por oneAPI, a síntese de alto nivel xerada a partir dos *kernels* de FPGA non foi competitiva para ningún dos casos avaliados. Unha observación a ter en conta é que para estes dispositivos o uso ou non de tamaños de problema coa forma $2^N$ é crucial. Isto débese a que a ferramenta de HLS xera os buses de datos de potencia de 2 en potencia de 2, facendo que cantidades de datos intermedias necesiten operacións adicionais de recheo ou aliñación, o cal incrementa substancialmente os tempos de transferencia.

No capítulo 4 avalíase a ferramenta de migración automática de CUDA a SYCL proporcionada por Intel, a cal se coñece como SYCLomatic, ou DPC++ Compatibility Tool (DPCT) na súa versión comercial ofrecida no conxunto de ferramentas de oneAPI. Para isto, migrouse a implementación en CUDA de Rodinia *benchmark suite*, unha coñecida colección de programas para avaliar o rendemento de aplicacións heteroxéneas. Foi necesario modificar levemente os códigos sen afectar á funcionalidade debido a que a última versión con soporte de Rodinia data de 2014, sendo necesarias estas modificacións para poder executalos utilizando unha versión moderna de CUDA. Ademais, introducíronse medicións estandarizadas para analizar os tempos de execución das distintas partes dos códigos e poder realizar comparativas precisas. Para estudar o grao de automatización ofrecido pola ferramenta, migráronse de forma automática todos os programas de Rodinia de CUDA a SYCL, obtendo un 93,24 % de éxito na mesma. A ferramenta tamén proporciona unha serie de mensaxes de alerta, ou *warnings*, con información sobre os procedementos realizados na migración,

detalles que deberían ser revisados polo programador ou simplemente avisos en relación ao rendemento, ou a incompatibilidades, entre outros. Os *warnings* obtidos tras a migración foron clasificados en distintos grupos segundo a súa función para poder avalialos, detectando que os máis frecuentes estaban relacionados con diferenzas entre as interfaces de programación de aplicacións ou Application Programming Interfaces (APIs) de CUDA e oneAPI (35,24 %), o rendemento (27,81 %) e o manexo de erros (17,41 %). Por outra banda, determinouse que a produción destes *warnings* non estaba relacionada directamente coa cantidade de liñas de código a migrar, polo que as dimensións do programa inicial non deberían condicionar o éxito da migración. A análise determinou que entran en xogo outros factores, probablemente relacionados coa dispoñibilidade de equivalencias apropiadas en SYCL ou a ausencia de rutinas non soportadas. En canto ao rendemento, realizáronse experimentos en dous nodos con GPUs de NVIDIA do supercomputador FT3 que constaron de 5 iteracións iniciais de quentamento, descartadas e utilizadas unicamente para inicializar compoñentes do sistema, seguidas de 10 medicións reais co fin de favorecer a fiabilidade dos resultados. Na maior parte dos *benchmarks* avaliados, as implementacións con CUDA e co SYCL migrado ofreceron tempos de execución similares nas dúas GPUs avaliadas. Do mesmo xeito, os valores de *performance portability* obtidos resultaron axeitados para plataformas GPU e foron lixeiramente superiores en SYCL. Para avaliar este aspecto empregáronse dúas métricas: $\Phi$, considerada nos últimos anos o estándar pola comunidade HPC para este tipo de estudos; e $\overline{\Phi}$, que penaliza de maneira menos agresiva os desbalanceos de rendemento entre plataformas, e que comezou a ser adoptada nos casos onde se calcula o *performance portability* a partir da eficiencia da aplicación, e non da arquitectura, como acontece neste estudo.

Entre as conclusións que se poden extraer tras a finalización deste traballo destácanse a clara tendencia á heteroxeneidade do *hardware* presente en todas as escalas dos sistemas informáticos, así como os retos computacionais que presenta este escenario, entre os que se atopa a estandarización das ferramentas de desenvolvemento para *software* heteroxéneo. Neste contexto, o estudo realizado no capítulo 3 permitiu confirmar a validez da implementación de SYCL realizada por Intel, oneAPI, para a programación deste tipo de aplicacións. De todas formas, recoméndase a implementación de *kernels* específicos para cada dispositivo no caso de querer priorizar o rendemento ademais da portabilidade. Este estudo tamén amosou os distintos graos de madurez que existen entre aceleradores, onde as FPGAs aínda non se atopan no mesmo punto que as GPUs en canto ao desenvolvemento de solucións de alto nivel. A mellora do ecosistema de ferramentas que rodea a estes dispositivos será fundamental para que experimenten unha democratización e adopción xeralizada, xa que simplemente polas súas características posúen un enorme potencial no futuro dos sistemas HPC heteroxéneos. Por outra parte, a aparición de tecnoloxías de migración automática, como a avaliada no capítulo 4, favorece a transición de modelos propietarios a modelos de programación multiplataforma.

# Contributions

This chapter presents the list of journal articles and conference presentations directly related to this thesis, as well as the software developed to carry out the studies conducted in the course of it.

- Articles in peer-reviewed journals

  S. R. Alcaraz[1], R. Laso[2], O. G. Lorenzo[1,3], D. L. Vilariño[3], T. F. Pena[1,3], and F. F. Rivera[1,3], "Assessing Intel OneAPI capabilities and cloud-performance for heterogeneous computing", The Journal of Supercomputing, vol. 80, pp. 13 295–13 316, 2024, issn: 1573-0484. doi: 10.1007/s11227-024-05958-5

  [1] Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela. Santiago de Compostela, Galicia, Spain.

  [2] Faculty of Informatics, Research Group for Parallel Computing, TU Wien. Vienna, Austria.

  [3] Departamento de Electrónica e Computación, Universidade de Santiago de Compostela. Santiago de Compostela, Galicia, Spain.

  - **Status:** Published (Open Access).
  - **Impact factor (JCR 2024):** 2.7, Q2.
  - **Category:** Computer science, theory and methods.
  - **Rank:** 51/147.
  - **PhD candidate contribution:** Conceptualisation of the work. Design, development, optimisation and validation of the software application. Design and conduct of the experiments. Analysis of the results. Preparation of the original manuscript.
  - **Related thesis content in:** Chapter 3.
  - **Reproduction rights:** "Reproduced with permission from Springer Nature". See Appendix C.

- Works at international conferences

S. R. Alcaraz[1], R. Laso[2], O. G. Lorenzo[1,3], D. L. Vilariño[3], T. F. Pena[1,3], and F. F. Rivera[1,3], "Performance evaluation of heterogeneous CPU+iGPU and CPU+FPGA schemes using Intel OneAPI on the cloud", in International Conference Computational and Mathematical Methods in Science and Engineering (CMMSE), Rota, Spain, 2023

[1] Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela. Santiago de Compostela, Galicia, Spain.

[2] Faculty of Informatics, Research Group for Parallel Computing, TU Wien. Vienna, Austria.

[3] Departamento de Electrónica e Computación, Universidade de Santiago de Compostela. Santiago de Compostela, Galicia, Spain.

- **Status:** Presented (without proceedings).
- **Impact factor:** Not ranked in GII-GRIN-SCIE (GGS) Conference Rating.
- **PhD candidate contribution:** Conceptualisation of the work. Design, development, optimisation and validation of the software application. Design and conduct of the experiments. Analysis of the results. Preparation of the original manuscript and public presentation of the work.
- **Related thesis content in:** Chapter 3.
- **Reproduction rights:** Not applicable.

S. R. Alcaraz[1], R. Laso[2], D. L. Vilariño[3], and F. F. Rivera[1,3], "SYCL for CPU+GPU Heterogeneous Computing: A Study on Integrated and Discrete GPUs", in 2025 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops), 2025, pp. 1–2. doi: 10.1109/CLUSTERWorkshops65972.2025.11164210.

[1] Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela. Santiago de Compostela, Galicia, Spain.

[2] Research Group for Scientific Computing, Faculty of Computer Science, University of Vienna. Vienna, Austria.

[3] Departamento de Electrónica e Computación, Universidade de Santiago de Compostela. Santiago de Compostela, Galicia, Spain.

- **Status:** Presented and published.
- **Impact factor:** CORE A, Class 2 in GII-GRIN-SCIE (GGS) Conference Rating.
- **PhD candidate contribution:** Conceptualisation of the work. Design, development, optimisation and validation of the software application. Design and conduct of the experiments. Analysis of the results. Preparation of the original manuscript and poster, and public presentation of the work.

  - **Related thesis content in:** Chapter 3.
  - **Reproduction rights:** "Reproduced with permission from © 2025 IEEE". See Appendix C.

- Developed software:

  - **ihp-sycl**: This project presents a microbenchmark designed to evaluate the capabilities and performance of Intel oneAPI for heterogeneous computing, using iterative problems as a case study. It solves two classical diffusion problems using the IHP [1] library for dynamic workload balancing across heterogeneous configurations, including CPU+GPU and CPU+FPGA setups. Available: `https://github.com/silviaralcaraz/ihp-sycl`.
  - **syclomatic-eval**: This project contains all the codes, scripts, and data necessary to evaluate the SYCLomatic tool, using the migration of the Rodinia benchmark suite from CUDA to SYCL as a case study. It contains the scripts needed to automate the initial analysis of the code to be migrated, migrate a given project, analyse the warnings obtained, run performance tests both locally and on a SLURM-based system, and analyse the timing results. Available: `https://github.com/silviaralcaraz/syclomatic-eval`.

# Chapter 1

# Introduction

*You can't stop change any more than you can stop the suns from setting.*
— Star Wars, *Episode I: The Phantom Menace (1999)*

From a historical perspective, the technological advances achieved since the beginning of modern computer science have set a precedent unseen in any other industry. Nowadays, all these continuous improvements and massive data generation by companies, research centres, and end-users have increased the computational requirements incredibly. Just to give some numbers, the Large Hadron Collider (LHC) at CERN [2] produced about 40 ZettaBytes (ZBs) of raw data in 2018, which is significantly more than the total data ever stored on Amazon S3, estimated at around 500 ExaBytes (EBs) as of 2021 [3]. In addition to this massive data generation, current trends such as Artificial Intelligence (AI) encourage specific and resource-intensive workloads. Concretely, in the field of Natural Language Processing (NLP), the size of the Large Language Models (LLMs) has been multiplying tenfold annually over the past years [4], resembling a new form of Moore's Law [5]. Managing these models not only entails optimisation techniques such as pruning and quantisation but also involves the creation of specialised hardware tailored to enhance the efficiency of both training and inference processes. Therefore, both in the past and in contemporary times, as the computational requirements of applications have grown, manufacturers have been continuously innovating to create more powerful computational devices equipped with larger and faster memory systems. The developer community, involving both academia and companies, has concentrated its efforts on developing techniques, strategies, and tools to exploit these resources in the most optimal way, maximising efficiency and performance and minimising power consumption. The development and

refinement of parallel computing techniques are crucial to this ecosystem. Parallel computing allows for the division of large problems into smaller ones, which can be processed simultaneously across multiple computing resources. This methodology is integral to High-Performance Computing (HPC), a field dedicated to leveraging parallel processing to achieve high computational speeds and manage large amounts of data effectively.

The origins of parallel computing can be traced back to the mid-20th century, when early computers operated on single-core processors, executing one instruction at a time. The 1960s saw the emergence of vector processors, able to execute a single instruction on multiple data simultaneously. According to Flynn's taxonomy [6], this is what we understand as Single Instruction, Multiple Data (SIMD), while in Multiple Instruction, Multiple Data (MIMD) architectures the processors operate independently both on different instructions and data. Around that time, networks also became robust and widely used, allowing distributed systems to gain popularity. Distributed computing harnesses the power of multiple connected machines to perform parallel tasks, effectively overcoming the physical and logical constraints of single-system architectures. Furthermore, developments such as Message Passing Interface (MPI) [7] helped standardise distributed parallel solutions, providing developers with tools to express communications in high-level languages.

The late 20th century was marked by efforts to shrink transistor sizes, enabling more cores and threads per processor to achieve higher computational throughput. However, there exists a consensus among industry experts and technologists that the post-Moore era started around the mid-2000s and early 2010s. This is a cause of the slowdown of the trends predicted by Moore's Law, which has driven advances in semiconductor technology for more than 40 years. Physical limitations related to the size of the transistors, the breakdown of Dennard Scaling [8], or the skyrocketing cost of making ever smaller designs are some of the reasons for this scenario. In response, the industry moved to multi-core processors during the 2000s, which consist of multiple processing cores placed on a single chip. The journey from single-core to multi-core processors marked a significant evolution in computing technology, enabling more efficient task processing through parallel execution within a single machine. It also emphasised the relevance of having a software stack that could utilise this hardware's parallelism, leading to advances in parallel programming languages and tools. Some examples are the introduction of libraries and frameworks for shared memory machines such as Open Multi-Processing (OpenMP) [9] or Intel Thread Building Blocks (TBB) [10].

At a certain point, the limitations of these homogeneous multi-core systems became apparent, paving the way for the advent of heterogeneous computing. This paradigm emerged as a solution that integrates diverse computational architectures within a single system, exploiting the unique strengths of each to optimise performance and power consumption. Currently, the most common scheme is the combination of Central Processing Units (CPUs) and Graphics Processing Units (GPUs). This was favoured

by the General-Purpose Computing on GPU (GPGPU) trend, where the extremely parallel architecture of GPUs started to be used as well for regular intensive parallel computational tasks, instead of just for graphics processing. The community and the development tools behind GPUs supported this adoption, especially for NVIDIA GPUs with the appearance of Compute Unified Device Architecture (CUDA) [11]. Nevertheless, there are other platforms to take into account, such as Field Programmable Gate Arrays (FPGAs). Their use as accelerators in heterogeneous systems is not yet as widespread due to the intrinsic difficulty of their programming and the lack of a tool ecosystem at the same level of maturity as that of other devices, such as GPUs. Their programming has traditionally relied on the use of Hardware Description Languages (HDLs), which required expertise in digital systems as well as low-level knowledge. Nowadays, High-Level Synthesis (HLS) tools have significantly simplified this task by providing automatic synthesis from high-level languages, such as C++, to HDL. The main advantage of FPGAs lies in the high flexibility that its reconfigurable capacity provides. This could be crucial to obtaining extra performance in certain types of problems, as well as more energy efficiency through the creation of ad-hoc applications. In fact, energy consumption has become a critical aspect and can be one of the keys to defining the role of FPGAs in modern HPC clusters. This issue becomes more severe the more powerful the system is, and it is necessary to consider that there are already supercomputers operating at the exascale level. One of the first was Aurora [12], developed by Intel and Cray for the Argonne National Laboratory, capable of reaching a performance of 1.01 Exa FLOPS (ExaFLOPS) while consuming 38.7 MegaWatt (MW) of power. Reflecting this concern about energy efficiency, there are two lists for classifying supercomputers worldwide: one is the Top500 [13], which includes the most powerful ones in terms of performance, while the Green500 [14] lists those that are most energy-efficient. Numerous studies in the literature, especially on homogeneous CPU clusters, monitor this aspect while proposing different mechanisms to achieve energy improvements. Currently, as already mentioned, the trend is for HPC clusters to be heterogeneous to better cover modern computational needs. Therefore, characterising the different components that affect the energy consumption of these systems is essential. The green computing trend focuses on designing systems that maximise energy efficiency while minimising environmental impact. These strategies not only help in reducing the carbon footprint but also in ensuring the sustainability of advancing computational technologies.

In the current landscape of computing, characterised by a progressive shift towards hardware heterogeneity, an increasingly diverse range of devices has become available. This includes accelerators already mentioned, such as GPUs and FPGAs, but also devices specifically designed for AI applications, such as Neural Processing Units (NPUs) and Tensor Processing Units (TPUs), among others. Within this environment, cross-platform programming models acquire particular importance, as they allow the execution of a single source code across different accelerators without the need for modifications. A representative case is the SYCL [15] programming

model, introduced by the Khronos Group [16], which aims to enable the development of portable applications while maintaining performance. It is worth noting that the adoption of cross-platform models is further facilitated by automatic migration tools, which translate applications originally written in vendor-specific frameworks into portable programming models and thereby reduce the entry barrier. In addition, it is essential to establish appropriate methods for assessing performance in such a heterogeneous context. The metric of performance portability addresses this need by measuring how efficiently and consistently a programme runs across a defined set of hardware platforms.

All these new architectures, advanced materials, and emerging technologies represent shifts away from traditional silicon-based transistor scaling. This means that progress in terms of computational capabilities continues, but it no longer focuses solely on transistor density as a driver of performance improvements. Nowadays, this process includes a broader array of novelties in software, system architecture, and computing paradigms, catalysing a new era in computing innovation.

Given the previous context, this work delves into the development and evaluation of portable heterogeneous computing solutions, assessing their performance on distributed HPC systems, such as cloud platforms and supercomputers. To this end, the performance of heterogeneous schemes incorporating different accelerators, specifically GPUs and FPGAs, was analysed in depth. Their advantages and drawbacks were also evaluated, together with their suitability for different problem types, while also considering the level of maturity of each device. The performance study was conducted on distributed computing systems, which provide access to a broader range of hardware and high-speed dedicated interconnects. Moreover, these environments represent realistic deployment scenarios for evaluating HPC-oriented applications. However, it must be noted that the utilisation of such systems also introduces limitations, such as restrictions on permissions for security reasons or a lack of access to certain low-level details. Concretely, experiments were carried out on a cloud environment, a supercomputer, and a high-end desktop system, whose specifications are detailed in the corresponding chapters. Finally, for the development of portable heterogeneous solutions integrating GPUs and FPGAs, a high-level cross-platform programming model was adopted using Intel oneAPI [17], Intel's implementation of SYCL. It uses Data Parallel C++ (DPC++) [18], which is based on C++17 and incorporates SYCL mechanisms to express parallelism and manage the memory, among other functionalities. This approach enabled applications to run seamlessly on both GPUs and FPGAs without requiring modifications to the source code. In the case of FPGAs, Intel oneAPI relies on the Intel Quartus HLS [19] tool to automatically translate high-level C++ code into HDL. In addition, the CUDA-to-SYCL migration tool provided by Intel, SYCLomatic [20], was evaluated with a focus on its degree of automation and the performance portability achieved on GPUs by comparing the native and migrated code.

## 1.1 Hypotheses and objectives

As was already introduced, the growing demand for computational performance and energy efficiency in scientific and industrial applications has driven the exploration of heterogeneous computing architectures. These systems combine different types of processing units with distinct strengths, such as CPUs, GPUs, and FPGAs. A central issue concerns the trade-offs between performance, portability, development effort, and device-specific constraints. Cross-platform development models and HLS tools for high-level FPGA implementation have emerged as promising approaches to alleviate the complexity of programming heterogeneous configurations.

At the same time, distributed computing systems provide access to a wide and powerful range of devices, enabling the development of heterogeneous applications oriented towards HPC. However, these systems also introduce certain challenges inherent to their design, which the programmer often has no control over, especially when using tools with a high degree of abstraction. These challenges include the way devices are interconnected through the underlying network, the lack of access to low-level details for security reasons, and the resource allocation policies, among others.

Considering this, the hypotheses of this PhD thesis can be formulated as follows:

| Hypothesis 1: |
| --- |
| Cross-platform programming models allow the development of performance-portable solutions across diverse architectures. |

| Hypothesis 2: |
| --- |
| HLS tools contribute to the democratisation of FPGAs as accelerators within heterogeneous systems, by improving their programmability and integration in software applications. |

| Hypothesis 3: |
| --- |
| Distributed computing systems provide suitable environments for the development and evaluation of portable heterogeneous solutions oriented towards HPC. |

Therefore, the main aim of this PhD thesis is to explore the development and evaluation of heterogeneous applications on distributed HPC systems from a high-level perspective, with a particular emphasis on portability. To investigate the mentioned hypotheses, the following specific objectives are established:

| Objective 1: |
| --- |
| Developing portable heterogeneous computing applications through cross-platform programming tools. |

| Objective 2: |
| --- |
| Identifying the advantages and drawbacks of each heterogeneous scheme, depending on the application and the current technological maturity of the respective devices. |

| Objective 3: |
| --- |
| Evaluating the performance and portability of heterogeneous applications on distributed HPC systems. |

## 1.2 General methodology

This work followed a classical scientific research methodology, beginning with a thorough study of the chosen fields of interest to identify existing challenges and define the scope of the investigation. In this case, the focus was placed on heterogeneous computing and distributed HPC systems, where the development of portable solutions was recognised as a promising approach given the current diversity of the accelerator ecosystem. The study then adopted an experimental approach as the basis for obtaining results and conclusions.

After the preliminary analysis, Intel oneAPI was selected as the development tool because it implements SYCL, a cross-platform programming model that enables the creation of portable heterogeneous solutions. Furthermore, GPUs and FPGAs were selected as target devices, given the widespread use and popularity of the first one and the promising characteristics of the second in the field of HPC. At the time this work was formalised, Intel oneAPI supported both platforms, reinforcing its selection. Moreover, the framework enabled the study of FPGA development from a high-level perspective, since it employs HLS to translate C++ code into HDL.

Concerning the experimental environments, distributed computing systems were chosen in order to access a broader and more diverse range of hardware, thus enabling more comprehensive evaluations. Moreover, these systems provided greater computational power than a domestic machine, making it possible to conduct studies oriented towards HPC. Specifically, experiments were performed on a cloud system and a supercomputer, in addition to a high-end desktop computer used for comparison. The chosen cloud system was the Intel Developer Cloud or DevCloud, now discontinued, since it provided a ready-to-use Intel oneAPI installation as well as access to a wide range of devices, including GPUs and FPGAs manufactured by Intel. Regarding the

supercomputer, the one used was the Finisterrae III (FT3) hosted by the Centro de Supercomputación de Galicia (CESGA), since the access was guaranteed to the research group, allowing experimentation on HPC-oriented NVIDIA GPUs.

Once the accelerators to be evaluated, the development framework and the evaluation environment had been defined, two experimental studies were proposed and carried out. On the one hand, heterogeneous CPU+GPU and CPU+FPGA configurations were evaluated, including both integrated and discrete devices from different vendors in the case of the GPUs. With this aim, two iterative problems were employed as case studies: one memory-bound and the other computationally intensive, in order to ensure a more representative assessment of real applications. Furthermore, a technique for dynamic load balancing between the host and the device was applied, and its impact on performance was analysed. On the other hand, a migration study was performed from CUDA to SYCL using the SYCLomatic tool, part of the Intel oneAPI toolkit under its commercial name DPC++ Compatibility Tool (DPCT) [21]. Given the importance of these tools in facilitating the transition from proprietary frameworks like CUDA to cross-platform programming models, the objective was to evaluate the level of automation of the tool as well as the performance and portability of the resulting code compared to the native implementation.

Regarding the performance evaluations, metrics such as execution time, scalability, Giga FLOPS (GFLOPS), and performance portability were measured using libraries, profiling tools, and scripts to automate data collection and calculations. Additionally, comparative analyses were conducted to evaluate the efficiency of the developed solutions under different hardware configurations or implementations.

In terms of reproducibility, this work includes all the information related to open-source tools, benchmarks, and data used, where applicable.

Finally, some of the most relevant results from this work have already been published in conferences and journals to share the findings obtained with the international scientific community.

## 1.3 Thesis outline

The rest of this manuscript is organised as follows:

- Chapter 2 provides an initial contextualisation of the two main topics addressed in this thesis. Section 2.1 includes a detailed explanation of heterogeneous computing, describing the evolution of GPUs and FPGAs, as well as their architectures and the development tools available for them. An in-depth discussion of cross-platform development tools and performance portability is also included. Section 2.2 offers an overview of distributed computing systems, explaining their advantages and challenges, the characteristics of the main paradigms oriented towards HPC, and an analysis of the increasing hardware heterogeneity together with some identified future trends.

- Chapter 3 presents a study on the capabilities and performance of SYCL for heterogeneous computing using the Intel oneAPI implementation. Iterative problems are considered case studies, applying a dynamic workload-balancing strategy between the host and the device in both CPU+GPU and CPU+FPGA configurations.
- Chapter 4 examines the degree of automation achieved by the SYCLomatic migration tool in translating a complete CUDA benchmark suite to SYCL, as well as the performance portability of the native and migrated applications.
- Chapter 5 summarises the conclusions obtained during the development of this work and identifies potential future research directions.

# Chapter 2

# Contextualisation

This chapter provides contextualisation for the two main topics related to the thesis: heterogeneous computing and distributed computing systems. Its purpose is to expand on the concepts and technologies introduced in Chapter 1 in order to facilitate a clearer understanding of the developments presented in Chapters 3 and 4.

## 2.1 Modern computing is heterogeneous

*Sequential programming is really hard,*
*and parallel programming is a step beyond that.*
— Andrew S. Tanenbaum, *Usenix Conference (2008)*

As introduced in the previous chapter, the slowdown in the rate at which processors enhance their capabilities has encouraged the pursuit of increased computational power by combining different devices into a single system. This idea is what we know nowadays as heterogeneous computing and although it provides multiple benefits in terms of performance and efficiency, multiple devices also mean multiple challenges to deal with. This section explains the key aspects of this paradigm, with a focus on the evolution and current state of Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs) as accelerators in heterogeneous schemes. Additionally, it discusses the emergence of multi-platform development tools for accelerators and introduces the concept of performance portability, detailing how it is rigorously assessed following established methodologies in the literature. This background is essential for future chapters.

### 2.1.1 Heterogeneous computing: possibilities and challenges

The heterogeneous computing paradigm consists of combining different devices to achieve higher performance and a more appropriate use of available resources by exploiting the advantages of each architecture. This is particularly relevant in the existing application environment, where workloads are becoming increasingly specific and, consequently, there is a need to utilise dedicated rather than general-purpose hardware to maximise performance and resource utilisation.

In the current landscape, the growing variety of accelerators available to address diverse computational tasks provides numerous options for designing heterogeneous systems, with the most common configuration being a combination of Central Processing Unit (CPU) and GPU. The fully parallel architecture of GPUs makes them ideal for problems that require high computational demand, as it enables multiple operations to be performed in parallel. Moreover, the ecosystem of tools surrounding them, along with their acceptance in both academia and industry, has contributed to their advantageous position. Section 2.1.2 provides an in-depth overview of their evolution, architectural characteristics, and the fundamentals of programming models for GPUs. Other devices which adopt a similarly highly specialised architectural approach to that of GPUs include Neural Processing Units (NPUs) and Tensor Processing Units (TPUs), which aim to meet the intensive and specific computational demands of Artificial Intelligence (AI) applications. Driven by the rapid growth of applications based on neural network models, these devices can be found in clusters, cloud infrastructures, and even Personal Computers (PCs). Although they are not evaluated in this work, it is important to highlight their recent emergence, as it emphasises the trend towards designing custom hardware to maximise performance for solving concrete tasks. In addition to GPUs, the other devices under study are FPGAs, with a focus on exploring the role of modern boards within heterogeneous schemes. Unlike CPUs and GPUs, FPGAs do not have a fixed architecture. Their reconfigurable capability provides them with high flexibility, a feature which can provide unique computational benefits depending on the application. In the last years, manufacturers have been significantly improving the internal complexity of FPGAs, while new tools have been introduced to simplify their programming. All these aspects are explored in Section 2.1.3, which focuses on the historical trajectory, architectural features, and the available tools for FPGA development. In this way, Section 2.1 aims to provide a grounded comparison of the progress of GPUs and FPGAs with a historical perspective, allowing for an understanding of the maturity level each has reached within the field of heterogeneous computing.

This increasing diversity and availability of accelerators allows for more tailored hardware selection to match specific workloads, thereby enhancing overall system performance. However, while heterogeneous computing improves suitability, it also introduces new challenges that have become subjects of study. In particular, specialised

computing devices or accelerators are capable of processing larger volumes of data simultaneously due to their architectural design. In contrast to having a greater area dedicated to performing operations in parallel, they typically possess less memory than, for instance, a general-purpose CPU. In recent years, manufacturers have been introducing improvements concerning both the quantity and capacity of memory in these devices; however, it is important to take into account that bandwidth may represent a potential bottleneck [22]. There are plenty of techniques used to mitigate this issue, such as overlapping computation and communication operations to reduce idle time and increase throughput [23–27]. Furthermore, the distribution of workload among the different devices is also a crucial task, as it aims to maximise concurrent use, avoiding unnecessary delays and load imbalances [28]. In addition, heterogeneous systems present a lack of standardisation and a fragmented ecosystem, as each device typically relies on a specific framework, sometimes even manufacturer-dependent. This often requires the coordination of multiple languages within a single project, making it difficult to maintain codebases or build portable applications. In response to these limitations in programmability and maintainability, cross-platform development tools for accelerators have emerged and are discussed in Section 2.1.4. Likewise, it is crucial to assess the performance delivered by different frameworks across various platforms, as well as to measure the differences that may exist between vendor-agnostic and proprietary solutions, among other considerations. The metric currently used for this purpose is performance portability, which is addressed in Section 2.1.5.

The challenges outlined above are further investigated in the following chapters through experimental analysis, as they represent critical aspects and active areas of research in the development of heterogeneous systems. Nevertheless, it is important to note that, due to the ongoing shift towards hardware heterogeneity in computing systems, this paradigm is continuously evolving. As a result, several emerging challenges remain beyond the scope of this discussion. Among these are the need to assess not only raw performance but also energy consumption, which has implications for energy-aware workload balancing strategies across devices [29, 30], thereby requiring the development of novel metrics. Additional issues include achieving efficient communication between heterogeneous components [31, 32], as well as ensuring security in multi-device environments [33]. These topics highlight the complexity and multidimensional nature of designing and managing modern heterogeneous computing platforms.

### 2.1.2 GPUs

Throughout the history of computing, the commitment to designing highly specialised hardware has not always led to good results. Nonetheless, GPUs represent a case of undeniable success, as well as complete acceptance and adoption in the market. The reasons for this can be summarised into four key factors: performance differen-

tiation, workload sufficiency, strong market demand, and the pursuit of mainstream adoption through affordable programming [34].

#### 2.1.2.1 GPU evolution

GPU evolution can be traced back to the early 1980s, when dedicated hardware for accelerating graphics computations began emerging. These pioneering chips performed fixed-function graphics tasks, meaning they were designed with dedicated circuits to handle specific graphics operations. The Geometry Engine (GE) [35], developed in 1981 at Stanford University and later commercialised by Silicon Graphics, Inc (SGI), marked a milestone in hardware-accelerated 3D graphics since it was one of the first dedicated graphics processors. The purpose of that chip was to accelerate three basic operations in computer graphics, including matrix transformations, clipping and mapping. During the 1980s and 1990s, other graphics accelerators emerged, such as: NEC µPD7220 [36], one of the first dedicated graphics display controllers used in early PCs graphics cards; IBM 8514/A (1987), considered the first widespread fixed-function graphics accelerator [37]; or SGI RealityEngine [38], a high-end graphics workstation capable of real-time 3D rendering. It may seem that the term Graphics Processing Unit was coined for these devices from the very beginning, but the truth is that it initially stood for *Geometric Processor Unit*. This was because they were mainly used to perform geometric computations such as Transform & Lighting (T&L), among others.

As technology advanced and the demand for graphics grew, these devices became capable of handling a wider range of operations, solidifying the term Graphics Processing Unit for GPU. In fact, the term was first used in 1999 by NVIDIA with the launch of the GeForce 256. Unlike previous graphic accelerators, it was able to process T&L operations in a single chip, reducing the CPU's workload in many applications. In 2002, the ATI Radeon 9700 [39] emerged, being the first fully programmable GPU and enabling the transition away from fixed-function or partial-programmable pipelines [40].

The ability of GPUs to perform high floating-point precision computations, combined with their recent programmable capabilities, made them an attractive platform to the scientific community. This led to the emergence of the General-Purpose Computing on GPU (GPGPU) trend, which refers to the use of a GPU to perform non-graphics computational tasks that would traditionally be handled by a CPU [41]. Even so, for this emerging trend to take hold, it was essential to provide a set of tools to develop programs for this platform. Programming GPUs for non-graphics tasks involved the use of graphics Application Programming Interfaces (APIs) such as OpenGL [42] or DirectX, which made the development process considerably complex. This need motivated NVIDIA to release Compute Unified Device Architecture (CUDA) [11] in 2007, a parallel programming model for coding algorithms on their GPUs. The CUDA programming model will be further described in Section 2.1.2.3, taken as a reference to

introduce important concepts related to GPU programming. The adoption of CUDA was facilitated by the creation of a huge ecosystem of libraries for scientific computations, such as CuBLAS [43], for efficient Basic Linear Algebra Subprograms (BLAS) operations. When the era of modern AI began, GPUs were quickly recognised as the most suitable hardware for training complex neural networks due to their highly parallel architecture. The turning point for this was the 2012 Large Scale Visual Recognition Challenge (ILSVRC) [44], where AlexNet [45] outperformed traditional Machine Learning (ML) approaches by a huge margin. This demonstrated that Deep Neural Networks (DNNs) trained on GPUs could achieve state-of-the-art performance in computer vision. In addition, on this occasion as well, the development of libraries such as cuDNN [46] and third-party frameworks like TensorFlow [47] and PyTorch [48] facilitated the adoption of GPUs as the hardware of choice for this type of application.

Recapping, GPUs have evolved from units that provided a fixed pipeline for graphics-related computations to the most parallel programmable devices on the market. They dominate in cutting-edge fields that can take advantage of their highly parallel hardware, such as Deep Learning (DL), and play a key role in the most powerful supercomputers today.

#### 2.1.2.2 GPU architecture

The primary performance difference between multicore CPUs and GPUs lies in the design philosophy of each one. Figure 2.1 summarises the main differences at the component level between the two devices. CPU design is optimised for sequential programs, employing sophisticated control logic to enable instructions from a single thread to execute in parallel or out of order. Large cache memories are used to reduce instruction and data access latencies in large applications. However, neither the control logic nor the cache memories directly contributes to computational performance. A similar situation arises with bandwidth, as CPUs must meet the requirements of the Operating System (OS), as well as the constraints of Input/Output (I/O) devices and applications. Driven by the rapid growth of the graphics industry, GPU manufacturers focused on maximising chip area and power allocation for floating-point operations. The core idea was to employ a large number of execution threads to continue processing while some threads remain stalled due to long-latency memory accesses, thereby reducing the control logic required for each thread. Additionally, the use of small cache memories to manage bandwidth demands ensures that threads accessing the same memory data do not all need to retrieve it from Dynamic RAM (DRAM), allowing for more chip area to be dedicated to floating-point computations [49].

GPUs primarily implement data-level parallelism through a Single Instruction, Multiple Data (SIMD) execution model, in which the same operation is applied simultaneously to multiple data elements. This paradigm is also found in vector processors, but in the specific case of GPUs, while the underlying hardware is SIMD-like, the

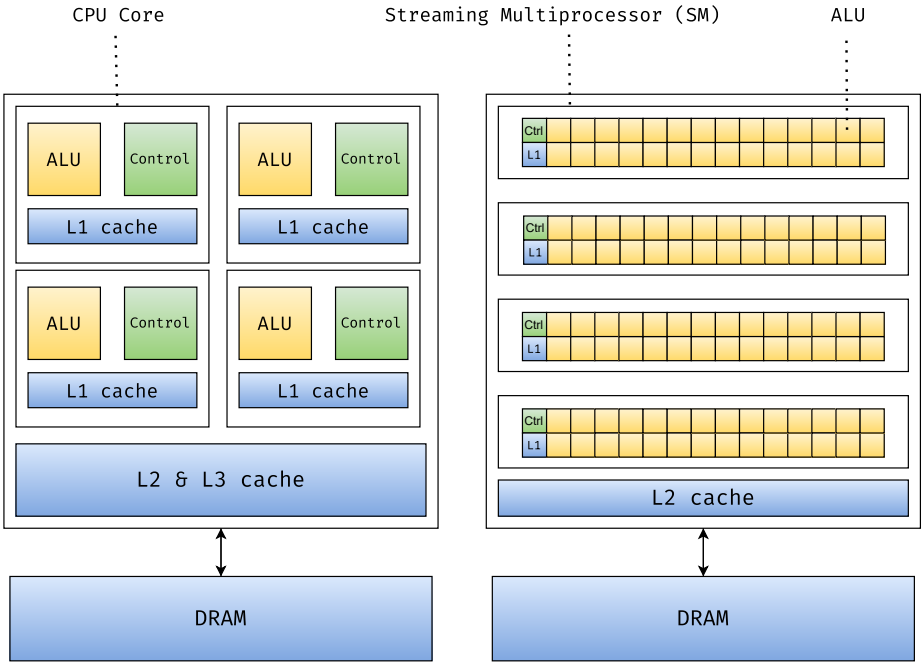



**Figure 2.1:** Comparison between CPU and GPU architectures.

programming model is more thread-oriented. As a result, GPUs are more accurately described as following a Single Instruction, Multiple Thread (SIMT) execution model. In this model, each thread executes the same instruction on different data but can follow its own control flow, providing greater flexibility compared to traditional SIMD architectures.

The core computational unit in these devices is the Streaming Multi-processor (SM), with each GPU consisting of hundreds or thousands of lightweight cores or threads designed for parallel execution. Each SM integrates multiple processing elements to enable high-throughput computations. Threads are grouped into **thread blocks**, and multiple blocks are organised into a larger structure called **grid**. The execution space is defined by the grid, allowing the GPU to manage and distribute the threads across the available SMs. To scale parallelism further, threads are executed in groups, with hardware schedulers managing these groups in a lockstep fashion to ensure efficient processing and mitigate memory latency. While the name and size of these groups vary across programming models, the core concept remains consistent. This can be seen in Table 2.1, where the terminology used in CUDA is compared with

Heterogeneous-Compute Interface for Portability (HIP) [50], the equivalent for AMD GPUs, and Open Computing Language (OpenCL) [51], a well-known multi-platform vendor-agnostic framework.

**Table 2.1:** Thread grouping concepts across different GPU programming models. (†)The number of threads is hardware dependent.

| Model | Concept | Size (threads) | Hardware |
|---|---|---|---|
| CUDA | Warp | 32 | NVIDIA GPUs |
| HIP | Wavefront | 64 | AMD GPUs |
| OpenCL | Subgroup | 32 or 64† | Vendor-agnostic |

The memory hierarchy is a critical component of GPU architecture, and it is also specialised for high-throughput. The on-chip components are the **shared memory** and the **registers**, characterised by being small, low-latency, and fast memories. Each thread block assigned to an SM can share data through shared memory, enabling more efficient cooperation and a better exploitation of data locality, whereas registers provide the fastest private storage for thread-local variables. The following memories are at the off-chip level, and they are larger, but also present high-latency: **global memory**, a primary data storage accessible by all threads; **local memory**, stored in the global memory space, but private to individual threads; **constant memory** a cached and read-only memory space optimised for storing constant data; and **texture memory**, optimised for specific access patterns and types of data. The physical backing of these off-chip memory spaces is the DRAM, although more recent GPU architectures utilise High Bandwidth Memory (HBM) to provide higher throughput and lower energy consumption. Another notable advancement in recent GPU architectures is the introduction of the so-called Unified Memory (UM) abstraction, which provides a single virtual address space shared between the CPU and GPU. The diversity of these memory components can benefit certain types of applications; however, efficiently utilising them remains a key challenge for programmers and APIs.

This hierarchical model—spanning execution units and a tiered memory system—facilitates efficient mapping of computations to the underlying hardware, allowing for large-scale parallelism and making GPUs well-suited for compute-intensive applications.

#### 2.1.2.3 GPU programming models

Having described the main components of GPU architecture, the following discussion addresses the fundamental aspects of the GPGPU programming models. Several frameworks offer the mechanisms required for GPU programming. Among these solutions, there are proprietary tools developed by hardware manufacturers for their

respective GPUs, such as CUDA by NVIDIA and Metal [52] by Apple. In the case of ROCm/HIP by AMD, it provides a native solution for AMD GPUs, as well as mechanisms to be compiled using the CUDA platform and be executed on NVIDIA GPUs [53]. In contrast, OpenCL, introduced in Section 2.1.2.2, differs in that it is fully vendor-agnostic, offering support across a range of hardware platforms. Other alternatives are discussed in Section 2.1.4. Overall, the evolution of GPU development tools is increasingly moving toward greater abstraction, specialisation, and portability across platforms, driven by the demands of modern applications. Clear evidence of this trend is the open-source framework developed by OpenAI, Triton [54], which aims to simplify development compared to CUDA for building neural networks, currently supporting both NVIDIA and AMD GPUs.

To present the foundations of GPU programming, CUDA is employed as an example, as it is considered a representative and widely adopted solution. Although it is a framework specific to the development of NVIDIA devices, it reflects broader GPU programming paradigms, which typically follow comparable hierarchical structures and memory models.

#### Heterogeneous paradigm

CUDA is based on a heterogeneous computing paradigm in which computations are divided between a **host** and a **device**. The host typically refers to the CPU, which is responsible for memory management and coordination tasks. The device, in this case the GPU, executes highly parallel computations. The portion of code executed on the device is known as **kernel**. The CPU launches these kernel functions, and each thread on the device executes the same code, but operates on different data. In the context of CUDA, the keyword `__global__` is used to declare a function as a kernel. Furthermore, the execution configuration, typically including the number of blocks $N$ and the number of threads per block $n$, is specified in the kernel call with the syntax <<<$N, n$>>>.

#### Thread and memory management

Kernels are executed in parallel across the many threads of the GPU. As explained in Section 2.1.2.2, the smallest execution unit is the thread, which is mapped to a GPU core. Moreover, the **warp** is the fundamental scheduling unit, and all threads in a warp execute the same instruction in SIMT fashion. The GPU's scheduler manages their execution, ensuring that multiple warps can be processed concurrently to maximise throughput. The unique identifiers of threads, blocks, and grids are crucial for determining memory access patterns and data distribution. CUDA allows these execution structures to be defined in up to three dimensions, which is especially useful for mapping computations to multidimensional data domains. These indices can be accessed within the kernel using the `threadIdx`, `blockIdx`, `blockDim`, and `gridDim` built-in variables, which allow threads to compute their specific task based on their

unique position in the overall hierarchy. The scheme provided by Figure 2.2 illustrates the idea of this execution representation, assuming that $N$ is the number of blocks per grid in the x-dimension and $n$ is the number of threads per block.

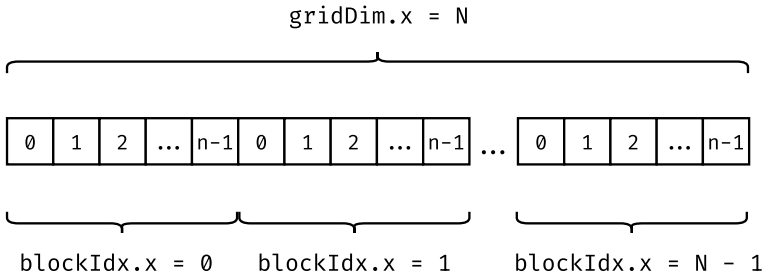



**Figure 2.2:** CUDA grid and block indexing for parallel execution.

An example of a CUDA kernel which performs a vector addition is depicted in Listing 2.1. In the example, line 4 computes the global thread index, which identifies the unique thread among all running threads. Line 11 shows the kernel call, indicating the execution configuration and the function parameters.

**Listing 2.1:** Vector addition example in CUDA.

```
1  template <typename T>
2  __global__ void vectorAdd(const T* A, const T* B, T* C, const int size)
3  {
4      int idx = threadIdx.x + blockIdx.x * blockDim.x;
5      if (idx < size) C[idx] = A[idx] + B[idx]; // Boundary check
6  }
7
8  int main(int argc, char *argv[])
9  {
10     [...]
11     vectorAdd<float><<<N, n>>>(d_A, d_B, d_C, size); // Kernel call
12     [...]
13 }
```

Regarding memory management, CUDA provides specific keywords to interact with memory spaces previously mentioned: `__device__` for global memory, `__shared__`, for shared memory, `__local__` for local memory, `__constant__` for constant memory, and `__texture__`, for texture memory. There is no dedicated keyword for registers; they are automatically allocated by the compiler for local variables within kernel functions, as is typical in heterogeneous computing frameworks. In the case of CUDA, the NVIDIA CUDA Compiler (NVCC) manages this allocation dynamically, optimising the use of registers while ensuring that they are used efficiently without explicit programmer intervention [55]. In addition to these keywords, the memory management

relies on a set of specific functions for allocation, transfer, and deallocation. To reserve memory on the device, the `cudaMalloc()` function is used, which allocates memory on the GPU's global memory. For transferring data between the host and the device, the `cudaMemcpy()` function is employed, which supports various memory copy operations, including copying data from the host to the device (`cudaMemcpyHostToDevice`) or from the device to the host (`cudaMemcpyDeviceToHost`). Finally, when memory is no longer needed, the `cudaFree()` function deallocates it on the device.

### 2.1.3 FPGAs

FPGAs are devices capable of reconfiguring part of their hardware after manufacturing to better meet the specific requirements of a given application. This capability enables enhanced performance in applications that benefit from a customised architecture. Moreover, the development of an ad-hoc solution leads to more efficient resource utilisation, thereby reducing energy consumption.

This subsection provides an overview of their historical evolution, architecture, and development ecosystem.

#### 2.1.3.1 FPGA evolution

The term reconfigurable computing refers to systems equipped with hardware programmability [56]. Essentially, this capability allows a single hardware platform to assume various configurations by adjusting its architecture to suit the specific requirements of different applications. This concept has a longstanding presence in computing history, with early references to the idea of achieving performance gains through restructurable computing components dating back to the 1960s [57].

In the late 1970s, a specific type of Programmable Logic Device (PLD) known as Programmable Array Logic (PAL) was introduced, being considered the first reconfigurable computing device [58]. These architectures featured a programmable array of *AND* gates feeding a fixed array of *OR* gates, providing greater flexibility than fixed-function logic, albeit with limited reconfigurability.

The decade of the 1980s was marked by the dominance of Application Specific ICs (ASICs) in digital logic design. These systems provide high capacity, performance, and low cost per unit, but they incur high Non-Recurring Engineering (NRE) costs, which refer to the one-time expense of researching, designing, developing, and testing a new product or system. For ASICs, a significant part of these costs stems from mask tooling, the complex and costly process of producing photolithographic masks for each chip layer. As a result, reconfigurable computing continued to advance as an alternative to reduce costs and enhance adaptability. This way, Complex PLDs (CPLDs) appeared in 1984, with Altera pioneering the first commercial one, the EP300. This

device featured erasable programmable logic and broader functionality compared to earlier devices. The major leap came in 1985 when Xilinx released the first commercially available FPGA, the XC2064 [59]. FPGAs allow for in-field reprogramming and multiple configurations, offering greater flexibility and adaptability compared to their precursors. Although FPGAs were initially slow, limited in capacity, and more expensive per unit, they did not incur significant NRE costs, as they did not require custom mask tooling or fabrication [60]. Due to this, they started to gain adoption for low-to-medium-volume applications, scenarios where ASICs had a prohibitive cost. Notably, these early FPGAs utilised Static RAM (SRAM)-based configuration, which allowed them to be reprogrammed multiple times, and antifuse technology, which provided a permanent programming solution.

FPGAs scaled following the Moore's law during the 1990s, doubling capacity and decreasing cost-per-function. As the design automation improved, large-scale FPGAs also became feasible. Moreover, SRAM-based FPGAs surpassed antifuse ones because they could take advantage of each new generation of manufacturing technology more quickly. These developments had further defined the market space for ASICs and FPGAs, relegating ASICs to applications requiring very high production volumes and specific performance. This occurred because the crossover point, the volume where ASICs become cheaper than FPGAs, continued to grow as the ASIC NRE cost increased. Even though chip manufacturing got cheaper per transistor, the cost of designing and preparing a custom ASIC kept rising sharply, especially because of masks, design complexity, and testing requirements.

In the 2000s, FPGAs evolved from simple logic fabric to comprehensive system-level platforms. Vendors integrated functional blocks such as Digital Signal Processing (DSP) units, memory controllers, and embedded processors, creating "Platform FPGAs" [60]. These new capabilities enabled them to efficiently handle computationally intensive tasks and support complex system functionalities, such as running Linux systems or managing network processes. The deceleration of Moore's Law in the latter part of the decade underscored the challenges associated with escalating hardware complexity, which in turn increased costs and design difficulties. In parallel, manufacturers began focusing on improving FPGA design productivity by developing Software Development Kits (SDKs) and Intellectual Property (IP) cores to simplify programmability and facilitate broader adoption across various applications. It is also worth noting that, during this decade and the previous one, alternative reconfigurable architectures were investigated to mitigate issues such as the high configuration overhead of fine-grained fabrics exemplified by FPGAs. Notably, Coarse-Grained Reconfigurable Arrays (CGRAs) emerged as an intermediate solution, sacrificing bit-level granularity in favour of word-level functional units to achieve a more balanced trade-off between flexibility, performance, and energy efficiency.

Over the last decade, FPGAs have increasingly been integrated into System on a Chip (SoC) architectures, which combine Programmable Logic (PL) with fixed hardware components, including embedded processors, peripheral interfaces, and other

system modules, on a single chip. The introduction of these components is motivated by the same reasons that led to the development of CGRAs, such as reducing configuration overhead, improving energy efficiency, and simplifying the mapping of data-parallel workloads. This convergence illustrates a broader trend towards heterogeneous reconfigurable platforms, where fine-grained programmability coexists with domain-specific efficiency. It is also important to note advancements in Electronic Design Automation (EDA) tools, which facilitate the optimisation of FPGA performance in terms of speed, power consumption, and latency, thus enabling their use in cutting-edge applications.

The result of this evolution, driven by advancements in semiconductor technology and shifting market demands, has determined the progress of the FPGAs, from prototyping technology or an alternative of ASICs when the cost is too high, to a feasible component in data centres or edge-computing systems.

#### 2.1.3.2 FPGA architecture

The reconfigurable capability of FPGAs lies in their architecture, which consists primarily of an array of Configurable Logic Blocks (CLBs) and a programmable interconnect used to link the CLBs with memory elements. CLBs, Programmable Logic Blocks (PLBs) or slices—depending on the vendor—are formed by several small elements like Look-Up Tables (LUTs), multiplexers, and Flip-Flops (FFs), and can be programmed to perform a wide range of logical functions. Coupled with the logic elements, the routing architecture comprises a network of programmable interconnects that enable the CLBs to be configured into numerous different functional arrangements, thus adding flexibility to the FPGA. This network includes a series of programmable switches and wiring channels, which can be electronically configured for routing signals among various parts of the FPGA. Each switch in the network can be opened or closed based on the configuration bits in the FPGA's bitstream. This enables connections or disconnections between the inputs and outputs of CLBs and the wiring channels, creating customised paths for data flow. The **bitstream** is the binary file which contains the configuration generated during the implementation and design process, which encodes all the information required to define the FPGA's logic and routing resources.

The way the FPGA's components are distributed could vary depending on the routing architecture. The traditional and most successful architecture in both academia and industry is the *island-style* [61], therefore, there are a lot of commercial FPGAs implementing it, such as the Altera Stratix II [62]. In this schema, CLBs are arranged in a grid with interconnects running horizontally and vertically around them, as shown in Figure 2.3. This design offers a balance between flexibility and complexity, suitable for most general applications. Alternatively, the *row-based* architecture aligns CLBs in rows separated by horizontal interconnect channels and is favoured for simpler, less

dense designs. More complex designs may utilise *hierarchical architectures* that group CLBs into larger blocks to manage connectivity and routing more efficiently at higher levels of integration. Each architecture type offers specific benefits, whether it is enhancing computational density, optimising power efficiency, or simplifying the design and routing methodologies.

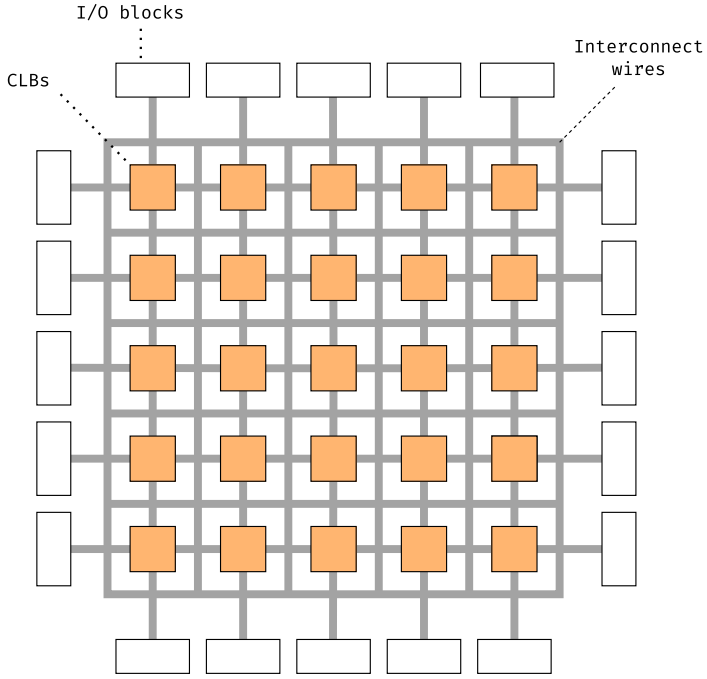



**Figure 2.3:** Basic FPGA *island-style* architecture.

As commented in Section 2.1.3.1, from the 1990s onward, the number of transistors on the FPGAs rose as established by Moore's law. Hence, their complexity increased, as did the amount of "hard" resources they contained to perform specific tasks efficiently [63]. These included DSP units to solve arithmetic or word-level logical operations, Block RAMs (BRAMs), or even the on-chip Advanced RISC Machine (ARM) microprocessors that most of the FPGAs have nowadays. This architectural evolution has led to modern FPGAs being composed of two distinct subsystems: the PL and the Processing System (PS). Although the integration of increasingly sophisticated PS components does not directly consume PL resources, it occupies chip area that could otherwise be used for additional reconfigurable logic. In any case, this trade-off

improves compatibility with other technologies, broadens applicability across diverse domains, and encourages the development of high-level programming tools.

#### 2.1.3.3 FPGA development ecosystem

A critical aspect associated with FPGAs, and in general to the field of digital logic design, is the inherent complexity involved in the development and verification of solutions for such devices. Various methodologies have been proposed to address this issue, offering different levels of abstraction from the final hardware implementation. The study by Sozzo et al. [64], which provides a thorough analysis of these methodologies, categorises them into Hardware Description Language (HDL), High-Level Synthesis (HLS), and Domain Specific Language (DSL). In this discussion, only the first two approaches will be examined and compared. Although both HLS and DSL provide a higher level of abstraction compared to HDL, a DSL——by definition——is tailored to a specific application domain, such as image processing or ML. HLS provides a more general approach to FPGA programming, making it better suited to the objectives of this thesis.

**HDLs**

The most traditional approach to develop FPGA solutions is the use of HDLs, which requires developers to possess low-level programming expertise and knowledge of digital systems. This significantly restricts the number of qualified programmers to develop such solutions, thereby hindering their adoption by professionals whose backgrounds are primarily in software engineering. HDLs remain the most widely used method, as they provide developers with a high degree of control over implementation, enabling design decisions to be made at a very low level. Consequently, solutions developed using HDLs tend to maximise both performance and resource utilisation. The principal HDLs still regarded as standards today were introduced in the 1980s, shortly after Mead and Conway [65] advocated for automation, abstraction, and the concept of compiling designs into silicon. These languages include Very High Speed Integrated Circuit HDL (VHDL) [66] and Verilog [67], as well as the latter's extended variant, SystemVerilog [68]. Despite the control that HDLs provide over the programmable logic, there are additional limitations, notably the substantial effort and time required for the verification of such applications, which limit their usability. This becomes clearer when examining the typical stages involved in the FPGA design process, illustrated in Figure 2.4 and described below:

1. **Specification and design**

   The design process begins with the definition of functional and performance requirements, including input and output specifications, timing constraints, and details of the target FPGA. These specifications are typically expressed using

HDLs, although high-level languages may also be employed, as will be discussed later.

2. **Simulation**

   In this step, the design is simulated to verify its logical correctness and functional behaviour. Simulation tools such as ModelSim [69] or Vivado Simulator, part of Vivado Design Suite [70], can be used to run testbenches and validate the design. This helps to catch functional errors early in the design process.

3. **Synthesis**

   This is one of the most critical stages. Synthesis involves translating the HDL code into a Register Transfer Level (RTL) description, which is a lower-level representation of the design that consists of registers, logic gates, and their interconnections. Synthesis tools such as Vivado [70] or Intel Quartus Prime [19] carry out this translation from high-level descriptions to RTL, and subsequently to a gate-level netlist, which serves as the final hardware representation.

4. **Place and route**

   In this step, the components described in the synthesised RTL netlist are mapped onto the physical resources of the target FPGA. The design tool determines the placement of elements (e. g., logic blocks, registers) and establishes the routing of interconnections based on the architecture of the specific device.

5. **Timing analysis**

   This process verifies whether the implemented design satisfies the specified timing constraints, such as setup and hold times, as well as signal propagation delays. The critical path, that is, the longest delay within the design, is also identified. If the design fails to meet these requirements, modifications must be made, revisiting the placement and routing process.

6. **Bitstream generation**

   Finally, the design configuration is generated into a binary bitstream file, previously introduced in Section 2.1.3.2. This file is the end product of the design flow and is used to configure the FPGA at runtime.


**Figure 2.4:** Steps in a common FPGA design workflow.

The low-level nature of HDL could make the verification process particularly time-consuming. It requires extensive testbenches and multiple simulation iterations to

ensure correctness, often involving checking every edge case of the design. Additionally, the synthesis and timing analysis stages can entail considerable time, especially for complex designs. Any timing violations or hardware resource inefficiencies discovered at these stages necessitate repeated adjustments to the HDL code, inherently arid and verbose, followed by re-synthesis and re-routing. As a result, the verification process becomes an iterative cycle of refinement and validation, substantially extending the overall design timeline.

### HLS tools

HLS has emerged as an alternative to address the HDL bottlenecks mentioned, and to improve productivity. The idea is to automatically generate efficient and reliable RTL code from high-level specifications (e. g., C, C++, or System C), thereby eliminating manual coding that may introduce errors and accelerating the typically drawn-out and iterative design cycle [71]. This approach automates several low-level hardware design tasks, including: the insertion of registers to break long combinational paths, helping to meet timing requirements and achieve the target clock frequency; the generation of standardised interfaces for system-level communication; the mapping of data to appropriate storage elements in order to balance resource utilisation and bandwidth; or the translation of computations into optimised logic elements.

Notwithstanding, the use of HLS tools implies additional steps in the FPGA design workflow, specifically for converting high-level code to HDL. The programmer is relieved from manually writing low-level code, but its quality depends on the tool's automatic translation, which may not be optimal and could affect the performance of the result. Although HLS tools have improved significantly since their first generations [72], it is crucial to pay close attention to the input provided by the developer, as it directly influences the resulting hardware synthesis. In essence, the HLS process involves the following steps prior to HDL generation [73]:

1. **Code analysis**

   The high-level code provided is examined, generating a Data Flow Graph (DFG) where each node represents an operation. The connections in this graph provide information related to dependencies among the different operations in the code.

2. **Resource allocation**

   The operations identified in the DFG are mapped to specific hardware resources, which are annotated with details regarding timing and area. Each operation may have multiple possible hardware implementations, each with different trade-offs in terms of area, delay, and latency.

3. **Scheduling**

A Data Path State Machine (DPSM) is created to provide the timing information, determining in which clock cycle each particular operation of the DFG will be executed.

Therefore, the quality of the high-level implementation provided has a direct impact on the quality of the synthesis performed. This code must be written in a style that aligns with the principles of HLS, significantly different from conventional software development. It is also crucial to be aware of the limitations inherent in this design paradigm. For instance, dynamic memory allocation is generally unsupported, pointer usage is restricted, and features such as virtual functions and standard libraries are often either limited or unavailable. In addition, HLS tools typically offer a range of compilation options that enable various optimisations, targeting aspects such as latency reduction, resource utilisation, or energy efficiency. Moreover, HLS developers have access to *pragmas*, directives that guide the implementation process. They can be grouped depending on their function; there are *pragmas* for optimising kernels or loops, packing data, managing interfaces, or applying function inlining. While the syntax of these directives may vary across different tools, all of them offer similar functionalities for improving the design by exploiting the FPGA architecture. The overarching goal of FPGA optimisation is twofold: to improve throughput and to reduce latency. In this context, throughput refers to the rate at which the design can accept new inputs, typically quantified as the number of data items or operations processed per unit time. Besides, latency denotes the number of clock cycles, or the amount of time, it takes for a single data item to traverse the entire design, from its input to its output. Both metrics are governed by the critical path, since it determines the minimum possible clock period. Shortening the critical path through different techniques enables higher clock frequencies and fewer cycles per transaction, resulting in higher throughput and lower latency, respectively. As follows, two commonly used techniques performed by the use of *pragmas* are explained. In this case, Vitis HLS [74, 75] syntax is used, due to its widespread adoption and maturity as an HLS technology:

- `#pragma HLS unroll [factor=N]`

  It works analogously to the classical loop unrolling technique. The *pragma* instructs the compiler to unroll the loop, meaning it generates multiple parallel instances of the loop body to improve parallelism and performance. The optional `factor` parameter determines the number of loop iterations to be unrolled. For instance, in Listing 2.2, the original loop executes four iterations; with `factor=2`, the compiler unrolls two iterations per loop cycle, partially unrolling the loop. The expected behaviour is depicted in Listing 2.3.

**Listing 2.2:** Loop unrolling example using an HLS pragma.

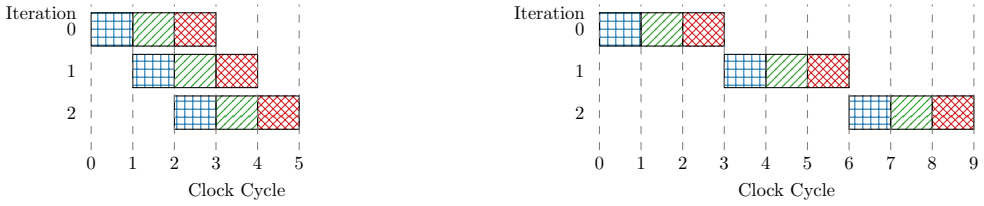
```
#pragma HLS unroll factor=2
for (int i = 0; i < 4; i++) {
    out[i] = in[i] + 1;
}
```

**Listing 2.3:** Expected loop unrolling behaviour in the proposed example.

```
out[0] = in[0] + 1; // Iter. 0
out[1] = in[1] + 1;
out[2] = in[2] + 1; // Iter. 1
out[3] = in[3] + 1;
```

- `#pragma HLS pipeline [II=<int>]`

  It enables loop or function pipelining, a technique that decomposes a task into a sequence of stages, each responsible for a portion of the overall computation. In a pipelined system, multiple data elements can be processed concurrently, with each residing in a different, independent stage. This allows new inputs to enter the pipeline depending on the Initiation Interval (II). Ideally, the value of II should be 1, meaning that a new operation begins every clock cycle. However, this is not always achievable due to data dependencies or limitations in the available hardware resources. If no II is explicitly specified, the compiler attempts to determine the minimum feasible II based on the characteristics of the design. FPGAs are particularly well-suited to pipelining, as they facilitate the creation of dedicated hardware for each pipeline stage. Unlike traditional processors, where multiple stages must share a fixed set of resources, FPGAs offer true parallelism and fine-grained control over both timing and resource allocation. This capability is illustrated in Figure 2.5.




(a) Pipelined execution (II=1)

(b) Sequential execution (CPU-like)

**Figure 2.5:** Comparison between (a) pipelined and (b) sequential execution for a 3-stage operation repeated over 3 iterations. Pipelining reduces total clock cycles by overlapping stages.

Since their apparition, HLS tools have experienced a continuous evolution, with an increasing number of options becoming available. Among these, Vivado/Vitis HLS—part of the Vitis Unified Software Platform [76]—stands out as the native solution for AMD Xilinx FPGAs. Given the strong market presence of this manufacturer in computing clusters and cloud infrastructures, it may be considered the most prominent commercial option nowadays. Intel has also developed native tools for its hardware, most notably the FPGA SDK for OpenCL, and the Intel oneAPI

Toolkit [17, 77], based on SYCL [15]. However, the first one was discontinued in 2022, and the latter has recently announced its deprecation for FPGA use following the transfer of Intel's FPGA division [78]. Other commercial HLS solutions include Catapult-HLS [79], LegUp [80], Stratus HLS [81], and XLS [82], with the latter two being vendor-agnostic. From the academic domain, alternatives such as Bambu [83] and Dynamatic [84, 85] have also been developed. All of the previously mentioned solutions are actively maintained at the time of writing this work, except Intel's tools, whose future remains uncertain due to recent commercial decisions and organisational changes [86].

It is important to remark that, beyond the tools themselves, a growing ecosystem of supporting libraries and frameworks has emerged to improve HLS programmability and extend its applicability. This is the case of PYNQ [87] framework, which provides an environment to integrate FPGA-accelerated functions or overlays directly in Python applications. Another example is HLS4ML [88, 89], which facilitates the deployment of ML models onto FPGAs, unless not all model types are supported. In addition, recent years have seen the emergence of libraries that provide abstractions to simplify HLS programming, such as hlslib [90].

The advancement of these tools, along with their ecosystem of libraries and frameworks, is fundamental in determining the role of FPGAs in mainstream heterogeneous computing scenarios [91]. While the progress of this device in the last decade in terms of hardware capabilities and development ecosystem is indisputable [92], it still faces some challenges in moving beyond niche sectors. Even high-level solutions such as HLS do not present a complete facility for the average software developer to create, maintain and deploy FPGA applications. This creates a lack of community and democratisation in environments such as data centres or clouds [93]. Advancing these tools toward greater abstraction and user-friendliness is essential for making FPGAs truly competitive in the current heterogeneous computing market.

### 2.1.4 Multi-platform developing tools for accelerators

The high-level development frameworks and tools previously mentioned in Section 2.1.2.3 for GPUs and in Section 2.1.3.3 for FPGAs address several challenges inherent to heterogeneous computing systems, such as programming complexity, device synchronisation, and efficient memory management. However, many of these solutions are languages or frameworks specifically designed for particular hardware architectures. This is the case of CUDA, which is tailored for NVIDIA GPUs. In the context of FPGAs, manufacturer dependency is typically introduced through the toolchain, which includes synthesisers, simulators, and place-and-route tools. Moreover, IP blocks, compiler directives (*pragmas*), and tool-specific extensions are often tied to specific vendors, further reinforcing this dependency.

The multi-platform (or cross-platform) development tools for accelerators have emerged in response to this increasingly heterogeneous hardware landscape, in which CPUs, GPUs, FPGAs, and other devices coexist. Their main objective is to facilitate code portability, understood as the ability of code to be successfully executed across multiple platforms. The promise behind these tools lies in enabling the development of a single code base that can run on diverse architectures without modification and, ideally, without sacrificing performance. Furthermore, this approach contributes to reduced maintenance costs by providing a unified syntax and promoting vendor independence. By abstracting the complexities of the underlying hardware, such tools allow developers to focus on the core logic of the application, while still achieving performance close to that of native code.

Table 2.2 presents an overview of programming models, frameworks and libraries that support multi-platform programming for accelerators. The collected information not only includes a breakdown of each tool's support for the devices studied in this work—CPUs, GPUs and FPGAs—but also a description according to their levels of abstraction. The existing compilation options for each are also included, as they serve to reflect the degree of integration, adoption, and maturity of each. Regarding the selection, toolkits targeting specific platforms, such as CUDA or Metal, have been excluded, as has Vitis Unified Software. Although Vitis is qualified as an Accelerator-Centric Synthesis (ACS) and enables programming of both the host and the device (specifically, the CPU and the FPGA), it is limited to Xilinx FPGAs, not being oriented to enhance portability. A brief description of each tool analysed is provided below:

- **OpenMP** [94]: API based on compiler directives, library routines, and environment variables for multi-platform shared-memory multiprocessing using C, C++ or Fortran as programming languages. Due to its maturity, a large number of compilers and tools implement it, including solutions by both universities and companies such as AMD, ARM, Intel, NVIDIA, among others. Its 4.0 specification introduces device offloading mechanisms, which allows to select on which device to run different tasks in heterogeneous computing environments. FPGA support has followed this path, with existing works that address program execution using OpenMP on single [95] or multi-FPGAs [96], the latter making use of OpenMP Cluster (OMPC) [97].

- **OpenCL** [51]: Open standard designed for cross-platform, low-level parallel programming of heterogeneous systems. It was the first widely adopted model to provide a common back-end for a broad range of platforms, including CPUs, GPUs, and FPGAs from several vendors, among other devices such as TPUs or DSPs. Its latest release, 3.0, introduces important improvements related to deployment flexibility. This version also maintains compatibility with FPGA workflows, as it is backwards-compatible. Nonetheless, in practice, most vendor

toolchains remain based on OpenCL 1.2 or 2.0, offering only a subset of the full 3.0 feature set due to hardware constraints and the limitations of HLS tools.

- **OpenACC** [98, 99]: A programming model that uses high-level compiler directives to express parallelism both at the code and compiler levels, enabling the construction of programs for multiple accelerators. To ensure portability, it defines an abstract model for accelerated computing, highlighting various levels of parallelism within a processor and a memory hierarchy with varying speeds and data addressing capabilities. Its primary aim is to ensure that OpenACC is not limited to a specific architecture, nor restricted to those widely available at the time, but remains adaptable to future devices. It also facilitates the offloading of both computations and data from a host to an accelerator device.

- **HIP** [50, 100]: C++ runtime API and kernel programming language that enables developers to build portable applications for heterogeneous systems, allowing them to run on CPUs, as well as AMD and NVIDIA GPUs from a unified source code. It is part of the AMD ROCm ecosystem, which also includes compilers, profilers, debugging tools, and libraries.

- **SYCL** [15]: An open-source, cross-platform abstraction to program heterogeneous solutions for different kinds of devices, such as CPUs, GPUs, and FPGAs. It is supported by the Khronos Group [16]. The main goal of SYCL is to provide a unified heterogeneous computing environment by allowing different hardware platforms to be programmed within a single source code using modern ISO C++ and APIs. Its latest specification [101] introduces notable enhancements such as the inclusion of Unified Shared Memory (USM), parallel reductions, work-group and sub-group algorithms, Class Template Argument Deduction (CTAD) to facilitate template instantiation, simplified accessors, expanded interoperability, and improved atomic operations. In terms of interoperability, SYCL provides a high-level interface to target accelerators, abstracting the execution model. The first initiatives implemented SYCL on top of OpenCL, owing to its support across a wide range of devices. However, today there are maintained SYCL implementations that use other back-ends. Three of these implementations are described due to their relevance and relation to this tool's review.

    - **ComputeCpp** [102]: A heterogeneous parallel programming platform by Codeplay [103] that implements SYCL 1.2.1 specification [104]. Among their supported platforms are Intel CPUs, and GPUs from different vendors, including Intel, AMD, and NVIDIA [105]. It works with OpenCL for back-end execution, meaning that it leverages the OpenCL runtime to support multiple devices, while using the SYCL standard to provide a higher level of abstraction and ease of programming.

  - **Intel oneAPI** [17]: SYCL's implementation provided by Intel and with native support for its devices, mainly including CPUs, GPUs, and FPGAs. Although, as mentioned in Section 2.1.3.3, the current status of the FPGAs is uncertain. The programming model is built around Data Parallel C++ (DPC++), based on C++17, and its current back-end is Intel oneAPI Level Zero [106]. The Intel oneAPI toolkits also provide high-level libraries, compilers, and debugging tools specific to Intel hardware. A more in-depth description of this tool will be given in Section 3.2, as it is the tool used in the development detailed in Chapter 3.
  - **AdaptiveCpp:** [107] Formerly known as hipSYCL or OpenSYCL, this project started as a research initiative at the University of Heidelberg. It is the only implementation that has never been based on OpenCL underneath, but rather on the HIP clang front-end, with added support for SYCL constructs [108]. Additionally, it provides back-ends for CUDA, HIP, and OpenMP.

- **Kokkos** [109]: Set of C++ libraries for writing portable applications for the main HPC platforms. It has a consolidated support for CUDA, HIP, SYCL, HPX, Open Multi-Processing (OpenMP) and C++ threads back-ends. Kokkos is also part of a bigger ecosystem, which includes Kokkos Kernels, a library of optimised linear algebra and graph algorithms, and Kokkos tools, a set of software to debug and profile applications.

- **RAJA** [110, 111]: C++14 library of abstractions originally developed as an academic project at Lawrence Livermore National Laboratory (LLNL). Its two main objectives are to enable portability of existing algorithms and programming languages and to achieve performance comparable to that of commonly used programming models. The library offers mature support for a variety of back-ends, including sequential CPU execution, OpenMP and Intel Thread Building Blocks (TBB) for parallel CPU multithreading, and CUDA and HIP for NVIDIA and AMD GPUs, respectively. It also provides experimental support for SIMD CPU vectorisation, OpenMP target offloading, and SYCL.

- **Alpaka** [112, 113]: Header-only C++20 abstraction for developing accelerators. It supports different devices, comprising several CPU families (x86, ARM, RISC-V and Power 8+), and GPUs from distinct vendors (NVIDIA, AMD and Intel). It achieves cross-platform compatibility by supporting a large number of back-ends: sequential CPU execution, OpenMP blocks and threads 2.0+, C++ `std::thread`, TBB 2.2+, CUDA 12.0+, HIP 6.0+, and SYCL through oneAPI 2024.2+. The support for FPGAs uses Intel oneAPI as a back-end, and it is still experimental. The library follows a model that exploits task and data parallelism, as well as the memory hierarchy which is presented in current multicore architectures.

- **HPX** [114]: Open-source library for concurrency and parallelism written in C++, with a special focus on ensuring high system utilisation and scalability. It implements functionalities to develop parallel, concurrent, and distributed solutions, also supporting several integration methods for GPUs [115].

**Table 2.2:** Overview of programming models, frameworks and libraries for cross-platform development. (*) Experimental/Low maturity. (†) HIP can execute kernels on the host for testing, but CPU back-ends are not performance-oriented.

| Tool name | Released | Supported platforms | | | Abstraction | Compiler(s) |
|---|---|---|---|---|---|---|
| | | CPU | GPU | FPGA | | |
| Programming model or framework | | | | | | |
| OpenMP [94] | 1997 | ✓ | ✓ | * | Directives | Wide support (e. g., GCC, clang, flang, icpx, NVHPC, AMD AOCC/ROCm, MSVC, HP CCE) |
| OpenCL [51] | 2009 | ✓ | ✓ | ✓ | Low-level C API | Wide support (e. g., ocloc, xocc, clang/LLVM-based) |
| OpenACC [99] | 2011 | ✓ | ✓ | * | Directives | Wide support (e. g., GCC, NVHPC, HP CCE, OmniCompiler, OpenUH) |
| HIP [100] | 2016 | ✓† | ✓ | ✗ | C++ API | hipcc (based on clang/LLVM) |
| ComputeCpp [102] | 2016 | ✓ | ✓ | ✗ | SYCL 1.2.1 | compute++ (based on clang) |
| Intel oneAPI [17] | 2020 | ✓ | ✓ | ✓ | SYCL 2020 | icpx -fsycl (based on modern clang/LLVM) |
| AdaptiveCpp [107] | 2022 | ✓ | ✓ | ✗ | SYCL 2020 | acpp (based on modern clang/LLVM) |
| Library | | | | | | |
| Kokkos [109] | 2012 | ✓ | ✓ | ✗ | C++ lib | ≥ C++17 compatible |
| RAJA [110] | 2015 | ✓ | ✓ | ✗ | C++ lib | ≥ C++14 compatible |
| Alpaka [112] | 2016 | ✓ | ✓ | * | Header-only C++ | ≥ C++20 compatible, ≥ nvcc 12.0, ≥ hipcc 6.1 |
| HPX [114] | 2021 | ✓ | * | ✗ | C++ lib | ≥ g++9.0, ≥ clang 10.0, ≥ MSVC++2019 |

In conjunction with the rise of multi-platform development tools and as a reflection of the growing interest in achieving compatibility across platforms and frameworks, automatic migration tools have also emerged. Given the widespread adoption of CUDA for GPU programming, many of these tools use CUDA as the primary source for migration. Notable examples include Intel DPC++ Compatibility Tool (DPCT) [21], and its open-source version SYCLomatic [20], which translates CUDA code to DPC++. Another example is HIPify [116], which converts CUDA to HIP. An

earlier project along the same idea is Cu2CL [117, 118], which facilitates migration from CUDA to OpenCL. In addition to these tools, several projects focus on developing appropriate back-ends or plugins to enable cross-platform languages to operate on specific hardware devices. One such example is the set of plugins provided by Codeplay, which allow DPC++ code——originally developed for Intel platforms——to run on NVIDIA [119] and AMD GPUs [120]. In the context of FPGAs, a similar initiative is the SYCL implementation known as TriSYCL [121]. This experimental project, no longer actively maintained, aims to facilitate the successful execution of SYCL code on AMD Xilinx FPGAs.

### 2.1.5 Performance portability

Throughout Section 2.1, the wide variety of device architectures available in today's heterogeneous computing systems has been explored, as well as the broad spectrum of cross-platform and platform-specific frameworks and tools. Considering this complex landscape, the establishment of a standardised evaluation methodology and a unified performance metric becomes imperative to meaningfully compare the performance of a given algorithm across different programming environments and hardware platforms. The metric adopted for this purpose is referred to as performance portability, usually denoted as "PP". It has long been used within the field of computing, albeit without a clear formulation or consistent understanding of what it actually measures. Thus, over the years, different definitions have been proposed [122–126].

Although it has already been exposed to several criticisms, the most widely adopted metric by the HPC community is the one proposed by Pennycook et al. [127]. The authors define performance portability ($\mathbf{\Phi}$) as *"a measurement of an application's performance efficiency for a given problem that can be executed correctly on all platforms in a given set"*. Regarding the formalisation, their proposal refers to the harmonic mean of an application's performance efficiency measured over a range of platforms. It is defined as

$$\mathbf{\Phi}(a,p,\mathcal{H}) = \begin{cases} \dfrac{|\mathcal{H}|}{\displaystyle\sum_{i\in\mathcal{H}} \dfrac{1}{e_i(a,p)}} & \text{if } e_i(a,p) > 0 \quad \forall i \in \mathcal{H} \\ 0 & \text{otherwise,} \end{cases} \tag{2.1}$$

where $e_i(a,p)$ is the performance efficiency of application $a$ solving problem $p$ on platform $i$, and $\mathcal{H}$ represents the complete set of platforms. The special-case $\mathbf{\Phi} = 0$ if any $e_i = 0$ just indicates failure or zero throughput on at least one target platform, since this metric uses a zero-tolerance rule.

Moreover, Pennycock et al. introduced two ways of measuring $e_i(a,p)$, expressed as values between 0 and 1 or percentages: architectural efficiency and application effi-

ciency, denoted by Equations (2.2) and (2.3), respectively. The latter can be expressed either in terms of achieved throughput relative to the best-known implementation or in terms of execution times, just taking into account that they are inversely related measures of performance.

$$e_i^{\text{arch}} = \frac{\text{measured throughput on } i}{\text{peak theoretical throughput of } i} \tag{2.2}$$

$$e_i^{\text{app}} = \frac{\text{measured throughput on } i}{\text{best-known throughput on } i} \tag{2.3}$$

In fact, the aggressiveness of the zero-rule, which in many cases produces unrealistic or biased results, together with the lack of proportionality, mathematical consistency, and the difficulties associated with interpreting and applying the metric, are among the main criticisms it has received. Many of these concerns were subsequently addressed through clarifications and corrections by the original authors, who revisited the formulation and refined some of the initial design criteria [128].

Given this context, alternative formulations have been suggested, most notably those proposed by Marowka. The first formal criticisms [129, 130] claim that the use of the harmonic mean introduces theoretical flaws, interpretability issues, and biased results in practice. He proposed replacing the harmonic mean with the arithmetic mean of efficiencies, which satisfies proportionality, avoids the strict zeroing behaviour, and provides a more intuitive assessment of performance portability. Formally, Marowka defines his metric as

$$\overline{\Phi}(a, p, \mathcal{S}, \mathcal{H}) = \begin{cases} \dfrac{\sum_{i \in \mathcal{S}} e_i(a, p)}{|\mathcal{S}|} & \text{if } |\mathcal{S}| > 0 \\ 0 & \text{otherwise,} \end{cases} \tag{2.4}$$

where $\mathcal{S} \subseteq \mathcal{H}$ is the subset of platforms on which the application is supported, and $e_i(a, p)$ the efficiency of application $a$ solving problem $p$ on platform $i$. This formulation, denoted as $\overline{\Phi}$, directly computes the arithmetic mean of efficiencies across the supported platforms, thereby preserving proportionality with the sum of scores and yielding values that are easier to interpret in practice.

This line of critique was expanded in subsequent works. In 2023, Marowka conducted a systematic comparison between the harmonic and arithmetic mean formulations, exposing that while they converge under balanced scenarios, Pennycook's definition yields disproportionately low scores when performance is unbalanced [131]. More recently, Marowka has advocated for an approach that combines arithmetic, harmonic, and geometric means as part of a portability efficiency measure [132]. This debate echoes earlier discussions about the most reliable way of evaluating performance when benchmarking, most notably the classic argument by Fleming and Wallace [133],

which demonstrated that the arithmetic mean can be misleading when applied to normalised performance data (e. g., efficiency), whereas the geometric mean provides a more consistent basis for comparison.

Marowka's proposals, while not yet displacing Pennycook's definition as the baseline, are increasingly being discussed and adopted by the community as complementary perspectives that enhance interpretability and provide a broader view of performance portability. In particular, they are gaining relevance for evaluating application efficiency or analysing the performance portability of different programming models [134, 135].

### 2.1.6 Heterogeneous computing in perspective

Section 2.1 has introduced how modern computing has become intrinsically heterogeneous, combining general-purpose processors with highly specialised accelerators to address the demands of current applications. Although hardware diversity provides unprecedented computational capabilities, it simultaneously poses challenges related to programmability and the lack of standardisation. This is particularly evident in the case of FPGAs, which offer flexibility and energy efficiency yet still suffer from steep learning curves and a more limited maturity of high-level development tools compared with other accelerators.

The trajectory of heterogeneous computing reveals a dual trend: on the one hand, the proliferation of specialised hardware; on the other, the pursuit of unified abstractions to bridge this diversity. Looking ahead, part of the progress in the field will depend on the evolution of high-level tools, the consolidation of cross-platform programming models, and the adoption of performance portability metrics to objectively guide design choices.

## 2.2 Distributed computing in the heterogeneous era

*You know you have one—distributed system—when the crash of a computer you've never heard of stops you from getting any work done.*
— Leslie Lamport, *2013 Turing Award*

The widespread adoption of distributed computing systems has been propelled by the growing demand for high computational power and efficiency across various domains. Their critical role is mostly driven by the advantages these systems offer in terms of scalability, reliability, or resource utilisation. By spreading the heterogeneous schemes explained in Section 2.1 across distributed environments, the complexity of these architectures increases significantly; nevertheless, so does their potential. This section introduces the fundamental concepts of distributed computing, exploring its advantages and challenges, the prevailing computing paradigms tailored to High-Performance Computing (HPC), the hardware heterogeneity present in such systems, and the emerging trends shaping the future of the field.

### 2.2.1 Foundations of distributed computing systems

A distributed system can be defined as a collection of independent computers that appears to its users as a single coherent system [136]. The possibility of creating such infrastructures was fundamentally motivated by two major historical milestones. First, the development of powerful microprocessors enabled the evolution from the initial 8-bit machines to today's 64-bit architectures. Second, the introduction of computer networks in the late 1970s [137] allowed numerous machines to connect and share amounts of data efficiently among them. These interconnections could span Local-Area Networks (LANs), Wide-Area Networks (WANs), or heterogeneous configurations combining multiple technologies, depending on the geographical scope and the application requirements. By combining these technological advancements, it became possible to interconnect large numbers of machines, giving rise to distributed systems or computer networks. This shift marked a departure from the traditional centralised paradigm, where only a single processor or computer handled all the work.

Although there are various types of distributed systems, this subsection focuses exclusively on distributed computing systems, which are normally used for HPC tasks. Typically, these systems consist of several key components, which are schematically represented in Figure 2.6. First, there is a collection of **nodes**, which are the individual computing units within the network that provide the computational power. These nodes can be physical computers, virtual machines, containers, or microcontrollers. Depending on their function, they can be categorised into different roles,

such as login nodes that handle user interaction, compute nodes that execute processing workloads, or management and storage nodes responsible for orchestration and data handling. Each node operates independently and has its own computational resources and memory, but coordinates with others to complete tasks. These nodes are connected by a **communication network** that facilitates synchronisation and data sharing. Networks implement communication protocols, such as Transmission Control Protocol (TCP)/Internet Protocol (IP), which describe the way to exchange messages, synchronise actions or transfer data. Significantly, the effectiveness of the system depends on the reliability and speed of this network. The software layer providing a common framework for communication and resource management across the nodes is known as **middleware** [138]. It abstracts the complexities of the network and hardware, presenting a uniform environment to applications.

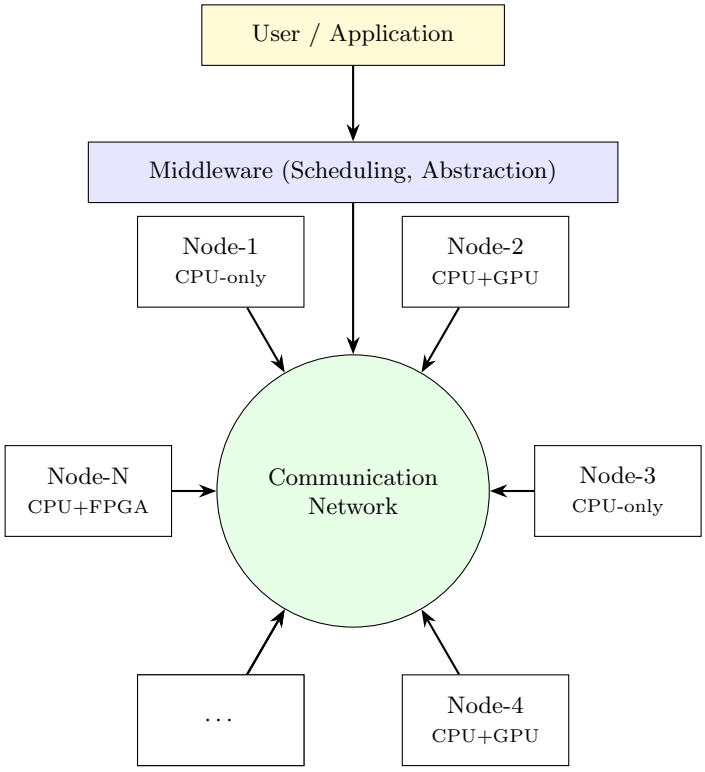



**Figure 2.6:** Simplification of a distributed computing system in which heterogeneous nodes are connected by a communication network.

With the current state of technology and the resources available to an average user, it is possible to acquire a machine with substantial storage and computational

power, or even to connect multiple such machines into a personal network. However, distributed computing systems offer key capabilities beyond simply networking powerful devices, among which accessibility, abstraction, security, and scalability stand out [136]. **Accessibility** refers to the ease with which users and programs can interact with remote resources as if they were local, while **abstraction** hides the details of distribution, such as the number of machines involved or their physical location. A particularly important aspect of abstraction is transparency, which allows the system to appear as a single coherent entity, despite being composed of multiple components. The different ways of implementing transparency contribute to this unified view and are essential for user convenience and developer productivity. Some types include access, location, concurrency, replication, and failure transparency. In addition, **security** becomes a critical concern in any distributed environment, especially when resources span administrative domains or are exposed over WANs. Ensuring confidentiality, integrity, and availability in such systems requires robust mechanisms for authentication, authorisation, encryption, and intrusion detection, often implemented across multiple layers of the system. Unlike in local computing environments, distributed systems must also handle threats arising from untrusted nodes and unsecured communication channels. Finally, **scalability** refers to the system's ability to proportionally accommodate growing workloads, larger datasets, or additional nodes without significant degradation in performance or manageability. This involves not only efficient resource allocation and load balancing but also architectural choices that avoid bottlenecks and enable elastic growth. Scalability is especially relevant in heterogeneous systems, where nodes may differ in processing power, memory capacity, or network bandwidth, requiring adaptive strategies to maximise overall system efficiency. Classic approaches to ensure scalability include replication, distribution, and caching strategies, not only to balance load and improve responsiveness, but also to ensure reliability and consistent performance as the system grows [139].

Together, these characteristics constitute the foundation of modern distributed computing systems, enabling them to support increasingly complex, data-intensive, and task-critical applications across a wide range of domains.

#### 2.2.1.1 Advantages and challenges

The characteristics discussed above translate into several practical advantages for distributed computing systems. These include improved scalability, allowing the system to grow in capacity and performance; cost-effectiveness, through shared use of resources, particularly in environments where high-performance or specialised hardware is prohibitively expensive; fault tolerance, whereby system reliability is maintained despite individual node failures; and security, through the use of decentralised access control and encrypted communication, especially when systems are designed with layered protection mechanisms across nodes. Moreover, the capacity to handle increasingly large datasets and diverse workloads makes these systems well-suited to

high-performance and data-intensive applications.

Despite these advantages, distributed computing systems present several important challenges. These can be broadly classified into those related to system design and those affecting application development and execution. Design-oriented challenges include the architectural complexity inherent in coordinating distributed nodes, the difficulty of maintaining data consistency across replicas, and the latency introduced by communication over networks. Furthermore, resource assignment policies are crucial to ensure fairness and efficiency across users and tasks. These policies are typically implemented through scheduling techniques that determine how and when resources are allocated. The choice of a scheduling strategy, such as priority-based or fair-share approaches, may significantly influence system throughput and user experience. In distributed systems, the coordination and timing of resource allocation are especially important, becoming an active area of study and interest [140, 141]. On the other hand, application-oriented challenges arise during development, deployment, and performance evaluation. Users often have limited access to low-level system details due to security and isolation requirements. These limitations may include restricted access to hardware counters or execution traces, which can constrain performance analysis and debugging. Additionally, developers are frequently responsible for implementing effective load-balancing strategies to maximise resource utilisation and ensure scalability. This task could be particularly complicated in heterogeneous systems where variability in hardware and communication connections further amplifies these trade-offs, as mentioned in Section 2.1.

Understanding the aforementioned complexities is essential when designing distributed computing systems, as well as when developing applications for them. Taking these intrinsic factors into account is key to maximising the advantages this architecture offers, not only in terms of computational raw power, but also in energy efficiency and cost reduction.

### 2.2.2 Distributed computing paradigms for HPC

Distributed computing encompasses a range of paradigms, each shaped by specific application domains, architectural models, and performance requirements. While these paradigms share the foundational principle of coordinating resources across multiple nodes, their design objectives and implementation strategies differ substantially. This subsection outlines the principal distributed computing paradigms, placing particular emphasis on those oriented towards HPC workloads. The classification adopted follows the three major paradigms identified by Hussain et al. [140]: cluster, grid, and cloud computing systems. However, cluster and cloud systems are examined in greater detail due to their use as experimental environments in this work. In addition, the specific case of supercomputing centres is considered as a representative class of high-end cluster systems. Other paradigms, such as edge and fog computing, are also briefly

discussed due to their growing potential to complement HPC systems within distributed processing pipelines, particularly in pre-processing, filtering, or decentralised data acquisition stages.

#### 2.2.2.1 Cluster computing

Cluster computing, also known as clustering, is currently one of the most established paradigms within high-performance distributed computing. Historically, its rise was driven by the convergence of four technological trends in the 1980s and 1990s: high-performance microprocessors, fast interconnects, standardised tools for distributed computing, and the growing computational demands of scientific workloads [142]. By definition, a cluster is a collection of interconnected, standalone computers that work collaboratively as a single, integrated computing resource. Modern clusters adopt a modular architecture, comprising compute nodes, storage, high-speed LANs interconnects, and resource management middleware. Traditionally, clusters were homogeneous, typically formed by CPU-only nodes. Nonetheless, their evolution also involves a shift towards heterogeneity, incorporating accelerators such as GPUs, forming environments capable of supporting a broader range of applications. Although still less common than GPUs, FPGAs have also been integrated into clusters, offering advantages in energy efficiency [143] and application-specific acceleration due to their reconfigurable architecture. Systems such as Axel [144] and ARUZ [145] demonstrate how FPGAs serve as complementary compute elements within heterogeneous HPC clusters [146]. This trend is further exemplified by initiatives like the Heterogeneous Accelerated Compute Clusters (HACC) program [147], which promotes the integration of diverse accelerators, including FPGAs, to enhance performance and flexibility in HPC environments.

A key element in cluster usability is the presence of resource management and job scheduling systems, such as Simple Linux Utility for Resources Management (SLURM) [148], which is widely adopted in both academic and industrial HPC environments. These tools provide functionality for job scheduling, resource allocation, and dependency-aware execution across the cluster. Other examples include Portable Batch System (PBS) [149], Torque [150], and Sun Grid Engine (SGE) [151], each with its own set of features and configuration models. Interaction with these systems typically occurs through the Command-Line Interface (CLI) that allows users to perform different tasks. While specific commands may vary slightly across systems, they often follow similar functionalities and semantics. Taking SLURM as reference, commonly used commands include `sbatch` for job submission, `squeue` for monitoring job queues, and `salloc` for launching interactive sessions[152].

From a programming perspective, cluster systems support a diverse range of parallelism models, each suited to different levels of the memory and execution hierarchy. These include message-passing paradigms, such as Message Passing Interface

(MPI), and shared-memory programming models, like OpenMP. Accelerator-based frameworks—including CUDA, OpenCL, and SYCL—enable the offloading of computation to specialised platforms, notably GPUs and FPGAs. This proliferation of programming models reflects the increasing architectural heterogeneity of modern clusters and the need for flexible abstractions to efficiently program heterogeneous many-core systems [153]. Furthermore, hybrid programming models (e. g., MPI+OpenMP or MPI+CUDA) are widely adopted in HPC clusters to efficiently exploit both inter-node and intra-node parallelism [154]. These combinations allow applications to scale across distributed nodes while fully leveraging on-node resources.

Nowadays, clusters remain the dominant architecture for medium to large-scale computational workloads due to their modularity, cost-efficiency, and broad programmability. They typically rely on commodity components and general-purpose configurations, which entail more relaxed performance constraints compared to specialised systems such as supercomputers, where peak computational capability is usually the primary, if not the sole, design goal.

**Supercomputer centres**

Supercomputers represent the high-end instantiation of the cluster computing paradigm. These facilities are designed to deliver extreme levels of computational performance by deploying large-scale, strongly integrated clusters composed of massive numbers of interconnected nodes, often equipped with specialised hardware. For this reason, they typically employ high-bandwidth and low-latency interconnects, hierarchical memory subsystems, and advanced cooling and power infrastructure to support tightly coupled workloads with long execution times.

Modern supercomputers frequently adopt heterogeneous architectures that combine CPUs with accelerators, typically GPUs, to improve throughput and energy efficiency. This landscape also includes systems incorporating FPGAs, such as Janus [155], an FPGA-based supercomputer developed for large-scale statistical physics simulations. More recent examples are *Cygnus* [156], which integrates Intel Stratix 10 FPGAs into hybrid CPU+FPGA nodes, and ESSPER [157], an experimental FPGA cluster developed to evaluate the potential of reconfigurable computing in high-performance scientific applications. It is worth mentioning that the latter is integrated alongside Fugaku [158], a supercomputer that ranked first in the TOP500 list from June 2020 to June 2022 and remains one of the most powerful systems in the world. Nonetheless, the adoption of heterogeneous FPGA-based HPC systems remains limited, primarily due to the complexity of hardware design and the lack of standard interfaces for interconnection, structure, and programming [146].

Regarding the software stack in supercomputers, it is tailored for large-scale parallelism and coordination. Therefore, it is typically combined with high-performance parallel file systems and advanced job schedulers, such as the ones previously men-

tioned, capable of managing massive job queues and complex dependency chains. These file systems enable efficient and concurrent access to large volumes of data across thousands of processes. One notable example is Lustre [159].

Supercomputing centres are generally operated by national research infrastructures, public laboratories, or industrial-scale Research and Development (R&D) organisations. They are essential for addressing "grand challenge problems" that exceed the capabilities of commodity hardware, such as climate modelling, genomics sequencing, materials simulation, and large-scale AI training [160], among many others. Prominent examples include Frontier [161], LUMI [162], and MareNostrum5 [163], which regularly appear among the top-ranked systems in the TOP500 list. Specifically, the Finisterrae III (FT3) supercomputer, hosted by the Centro de Supercomputación de Galicia (CESGA), is employed as an experimental environment in this work.

While supercomputers are architecturally based on clustered designs, their scale, integration, and engineering sophistication distinguish them from conventional computer clusters. Extreme performance is achieved not only through hardware resources but also via tightly coordinated software stacks, optimised communication layers, advanced scheduling policies, and dedicated access and usage models.

#### 2.2.2.2 Grid computing

Grid computing emerged in the late 1990s as a paradigm for enabling coordinated, scalable, and secure resource sharing across geographically distributed and administratively independent computing infrastructures [164]. While clusters are typically co-located and centrally managed, grids are inherently federated and decentralised, designed to integrate heterogeneous resources from multiple institutions [140]. The primary goal of grid computing is to provide access to distributed computational and data resources as a unified, virtual infrastructure. This is achieved through middleware frameworks that handle authentication, authorisation, data transfer, and job scheduling across heterogeneous platforms and administrative domains [140, 164]. These middleware layers hide the underlying system complexity and provide standardised interfaces to users and applications.

Grids have played an important role in advancing large-scale scientific projects that required distributed data processing and computational capabilities. One notable example is the Worldwide LHC Computing Grid (WLCG) [165], which supports data analysis for experiments at CERN.

#### 2.2.2.3 Cloud computing

Cloud computing is a distributed computing paradigm that enables ubiquitous, on-demand, scalable access to a shared pool of configurable computing resources (e. g.,

processing units, storage, or networking) through a service-oriented model. Its essential characteristics are self-service provisioning, broad network access, resource pooling, rapid elasticity, and measured service accounting [166]. It emerged in the mid-2000s as a flexible and elastic alternative to traditional infrastructures, supported by advances in virtualisation, web technologies, and data centre automation.

In contrast to clusters and grids, cloud platforms are designed around resource abstraction, dynamic provisioning, and user-centric access models. Resources are typically provisioned through Virtual Machines (VMs) or containers, orchestrated via middleware such as OpenStack [167], Kubernetes [168], or commercial platforms like Amazon Web Services (AWS) [169], Microsoft Azure [170], and Google Cloud Platform (GCP) [171]. Users interact with these platforms through APIs, portals, or Infrastructure as Code (IaC) interfaces, eliminating the need to manage the underlying hardware.

To support diverse use cases, cloud computing adopts three core service delivery models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS), each providing increasing levels of abstraction from the underlying infrastructure. **IaaS** offers virtualised compute, storage, and networking resources that can be configured by users, granting control over operating systems and applications while abstracting the physical hardware. For instance, Amazon Elastic Compute Cloud (EC2) [172] and Microsoft Azure Virtual Machines [173] exemplify IaaS platforms. **PaaS** provides a development environment with integrated tools and runtime support, allowing developers to focus on code without dealing with low-level infrastructure. A representative example is Google App Engine [174]. **SaaS** delivers fully managed software applications via thin clients such as web browsers; examples include Google Drive and Microsoft 365, where users access functionality without maintaining the underlying system. Complementing these service models, several organisational deployment models define how cloud infrastructure is managed and accessed: a **public cloud** is owned and operated by a third-party provider and made available to the general public over the Internet, as exemplified by AWS or GCP; a **private cloud** is dedicated to a single organisation, offering greater control and customisation, and may be managed internally or by a third party; a **community cloud** is shared by several organisations with aligned interests, such as regulatory compliance or collaborative research, and is often jointly managed; and a **hybrid cloud** integrates two or more cloud types to combine their respective advantages, supporting flexible workload placement and data portability across environments [175]. Figure 2.7 summarises the conceptual structure of cloud computing models, distinguishing between service and deployment perspectives discussed above.

From an HPC perspective, the cloud paradigm presents both opportunities and limitations. On the one hand, cloud elasticity and the pay-as-you-go model are well suited for embarrassingly parallel tasks, large-scale data analytics, and workload bursts. On the other hand, virtualisation overheads, unpredictable network latencies, and shared resource contention make cloud environments less suitable for tightly-coupled, low-

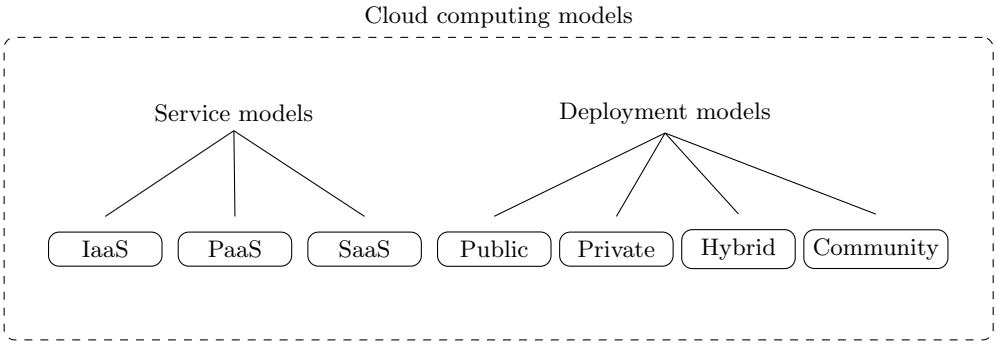



**Figure 2.7:** Cloud computing models, distinguishing between service delivery and deployment models.

latency HPC applications [140]. To mitigate these challenges, recent developments in cloud infrastructure have introduced specialised HPC instances with low-latency interconnects (e. g., Elastic Fabric Adapter (EFA) [176] in AWS) and native support for development tools, such as MPI and acceleration frameworks for GPUs. Complementary strategies involve the previously discussed hybrid clouds, increasingly adopted in HPC workflows. They combine the scalability of public clouds with the control and low-latency capabilities of private infrastructures, allowing for strategic workload partitioning, where tightly-coupled or security-sensitive applications remain on-premise, while embarrassingly parallel or burst workloads leverage elastic cloud capacity. In addition, emerging models such as fog and edge computing address some of these limitations by relocating computation closer to data sources. For this reason, they are further examined in Section 2.2.2.4.

Another growing trend in cloud evolution is the integration of reconfigurable computing resources, particularly FPGAs. These "reconfigurable clouds" support adaptable hardware acceleration for tasks such as DL, genomics, or network processing [93]. As will be discussed in Section 2.2.3, some cloud providers have deployed FPGAs at scale, offering them either directly to customers or as internal acceleration engines. Notably, FPGAs can process data directly from the network, enabling low-latency acceleration of services such as web search or host networking [93]. This architectural integration allows providers to transparently enhance performance for users, even without their explicit awareness. A prominent example of this design is the use of programmable Network Interface Cards (NICs) or SmartNICs [177–179], which integrate reconfigurable logic achieved through embedded compute units, such as FPGAs. They reduce CPU overhead, provide lower latency, and improve throughput and energy efficiency by processing data directly at the network edge. This capability makes them particularly well suited for modern data-intensive cloud workloads, like packet filtering, load balancing, or encryption, for example.

In summary, cloud computing constitutes a flexible and essential component of the HPC ecosystem, particularly in scenarios involving burst workloads, data-centric pipelines, and complex workflow orchestration. Furthermore, it provides on-demand availability of computational resources through a pay-as-you-go model, enhancing both accessibility and cost-efficiency.

#### 2.2.2.4 Emerging HPC paradigms: edge and fog computing

Although not traditionally associated with HPC, edge and fog computing represent paradigms which extend the scope of distributed computing and support novel workloads. They have gained increasing relevance in response to the proliferation of Internet of Things (IoT) and data-intensive applications with real-time constraints. In contrast to the centralised architecture of cloud systems, edge and fog follow hierarchical, decentralised models that prioritise responsiveness, bandwidth efficiency, location awareness, and local autonomy over raw computational throughput [180].

The core idea behind edge computing is to process data close to its source, such as sensors, embedded devices, or mobile platforms. By relocating computation nearer to end devices, this model significantly reduces latency, conserves bandwidth, and enables real-time processing. These systems typically operate under energy, compute, and connectivity constraints, which necessitate lightweight and adaptive algorithms. Despite these limitations, edge computing supports context-awareness, enhances privacy by keeping data local, and provides scalability by distributing workloads across a heterogeneous set of devices [181]. In contrast, fog computing introduces an intermediate layer between edge devices and cloud or data centre resources, providing intermediate processing, storage, and control capabilities. This layer is composed of so-called "fog nodes", like gateways, routers, or embedded servers, which have greater capacity and connectivity than edge devices. Hence, fog computing enables the development of large-scale sensor networks, promotes service elasticity, and improves resilience in heterogeneous and dynamic deployments [182].

Although both paradigms aim to reduce latency and bandwidth usage by relocating computation, their architectural focus differs. Edge computing operates normally on resource-constrained devices, while fog computing relies on intermediary nodes with higher processing capacity that can coordinate multiple devices and manage shared workloads. As a result, resource contention is generally more significant in edge environments [183, 184].

Due to their respective characteristics, edge and fog computing are well-suited for latency-sensitive or bandwidth-constrained applications in which offloading all computation to centralised infrastructures is impractical. Common use cases include autonomous systems, industrial IoT, smart cities, and augmented reality, among others. Although not designed for tightly coupled parallel workloads, both paradigms are increasingly integrated into hybrid distributed architectures. They often serve as front-

end layers that perform initial tasks, such as data filtering or aggregation, thereby reducing the computational and bandwidth burden on back-end HPC infrastructures. This layered approach is depicted in Figure 2.8, which exemplifies the complementary roles of edge, fog, and cloud resources in modern HPC workflows.

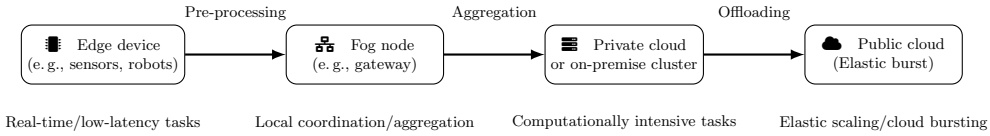


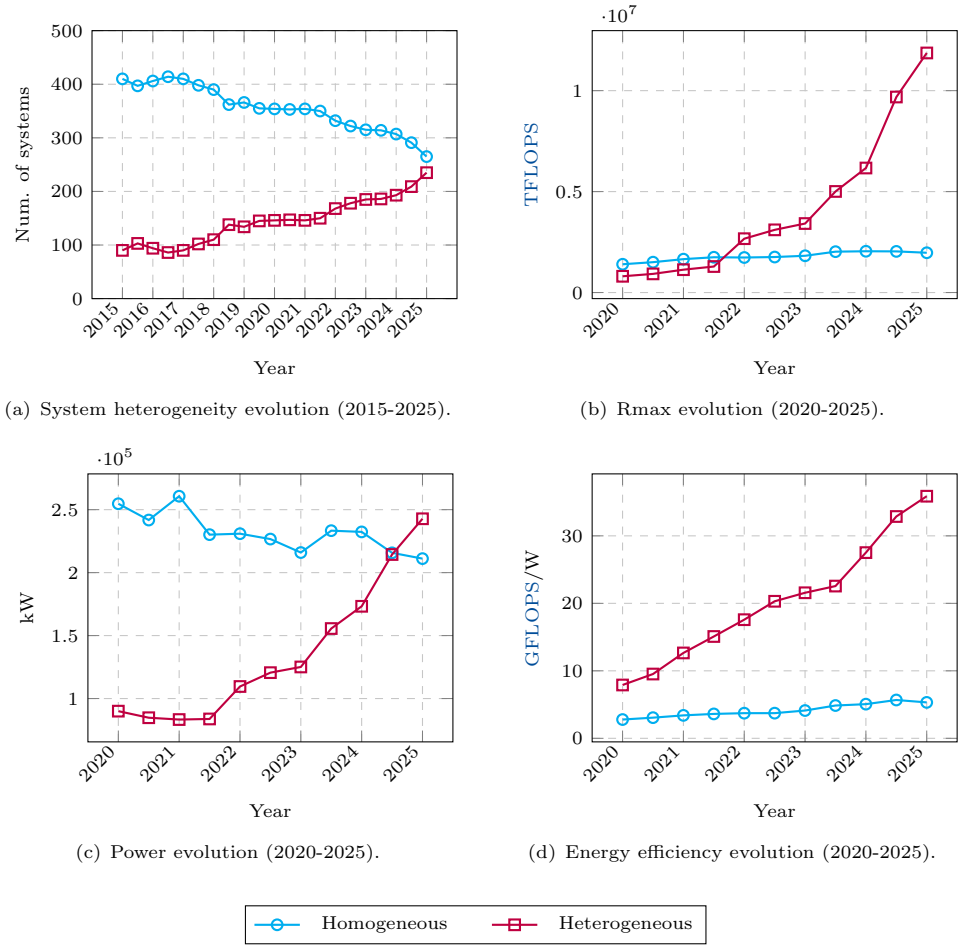

**Figure 2.8:** Layered integration of edge, fog, and cloud resources. In this example, edge and fog nodes perform low-latency processing and local coordination, offloading selected tasks to specialised HPC back-ends.

### 2.2.3 Heterogeneity at scale: supercomputers and clouds

As highlighted previously, modern computing has progressively shifted towards hardware heterogeneity in response to the growing computational demands of contemporary applications. This trend spans all computational scales, from personal systems to large-scale distributed infrastructures. This subsection analyses how heterogeneity manifests in major distributed systems, with a focus on those targeting HPC workloads. The discussion is structured around two key domains in which hardware heterogeneity is analysed: the evolution of supercomputers and the architectural diversity of current cloud computing platforms.

To examine supercomputing centres, a study was conducted using the publicly available reports from the Top500 website [13], which compiles data on the world's 500 most powerful supercomputers twice a year, in November and June. Specifically, Figure 2.9(a) illustrates the evolution of supercomputers over the last decade. It can be observed that in 2015, only about 100 systems featured an accelerator or coprocessor; however, heterogeneity has grown steadily over time. By 2025, the number of homogeneous and heterogeneous systems becomes nearly equal, 265 and 235 systems, respectively. Within the heterogeneous category, the widespread adoption of NVIDIA GPUs, present in 202 systems, reflects the prevailing role of NVIDIA-based acceleration in contemporary HPC architectures.

This hardware divergence has implications not only for raw performance but also for power consumption and energy efficiency. Figures 2.9(b), 2.9(c) and 2.9(d) present aggregate metrics for homogeneous and heterogeneous systems over the past five years (2020–2025), a sufficient period to reflect relevant trends given the rapid pace of hardware innovation. Figure 2.9(b) shows the evolution of Rmax, the maximum measured performance of a supercomputer based on the LINPACK benchmark [185], which reflects its peak computational throughput. From 2022 onward, this figure reveals




(a) System heterogeneity evolution (2015-2025).

(b) Rmax evolution (2020-2025).

(c) Power evolution (2020-2025).

(d) Energy efficiency evolution (2020-2025).

**Figure 2.9:** Comparison of heterogeneous and homogeneous systems throughout the evolution of the TOP500 rankings.

a consistent performance advantage for heterogeneous systems. While this partially stems from their increased presence, it also reflects the inherent scalability advantages of accelerator-based architectures, especially as general-purpose processors approach physical and thermal limits. Regarding power consumption, illustrated in Figure 2.9(c), the trend until the first half of 2024 aligns with the belief that accelerators offer better energy utilisation compared to CPUs. Nonetheless, the results from 2025 indicate that power consumption by heterogeneous systems surpasses that of homogeneous systems. This is partly due to the increased number of these systems, but also because of the vast number of accelerators included in current supercomputers to achieve their top computational power. These results underscore the growing challenge of balancing compute performance with sustainable energy consumption. Accordingly, improving energy efficiency, rather than maximising raw performance alone, has become a central concern, particularly as HPC systems increasingly contribute to global carbon emissions. Finally, despite the mentioned increase in energy consumption by heterogeneous systems, Figure 2.9(d) demonstrates that these systems continue to present higher energy efficiency.

Concerning the main cloud computing platforms, they also exhibit substantial heterogeneity:

- **AWS** [169] enables direct reservation of CPUs (Intel Xeon/Core, AMD EPYC, AWS ARM-based Graviton), GPUs by NVIDIA (instances G4-G6 and P4-P6), and AMD (instance G4ad), as well as AI accelerators (Inferentia and Trainium by AWS, Gaudi by Intel, and Qualcomm AI 100). It also supports fully customisable FPGA instances (EC2 F2 and VT1) featuring AMD Xilinx boards (families Virtex and Alveo, respectively), accessible via the FPGA Developer Amazon Machine Image (AMI) and Amazon FPGA Image workflows. Although networking is accelerated via the ENA SmartNIC (Nitro), this component is fixed-function and not user-programmable.

- **Microsoft Azure** [170] permits reservation of CPUs (Intel Xeon, AMD EPYC, and ARM-based Microsoft Cobalt and Ampere Altra), NVIDIA and AMD GPUs (families NC/ND/NG/NV), and fully-programmable FPGAs (NP and NM families, with AMD Xilinx Alveo cards). Regarding AI accelerators, Azure does not currently offer instances to book with NPUs. In the same way, Azure Boost Data Processing Units (DPUs) are only integrated into Azure datacenter servers to offload processing tasks like networking, storage, security, and data movement.

- **GCP** [171] provides reservable CPUs (Intel Xeon, AMD EPYC, ARM-based YiTian 710 and Ampere Altra) and NVIDIA GPUs (V100/A100). It offers TPUs (v2-v5, and Trillium), which developers can program via supported ML or DL frameworks. However, GCP does not provide user-programmable FPGA instances.

- **Alibaba Cloud** [186] supports CPUs (Xeon, EPYC, and some ARM-based), NVIDIA GPUs, and in-built NPUs available via Elastic Cloud Service (ECS) instances and its Alibaba Cloud AI Acceleration Engine (AIACC) framework [187]. It offers FPGA-as-a-Service through ECS (f1-f3 instances with AMD Xilinx and Intel devices) [188].
- **Intel Tiber AI Cloud** [189] is the evolution of the deprecated Intel DevCloud, centred on offering access to Intel's hardware, but with an orientation to AI workloads. It allows reservation of CPUs (e. g., Xeon and Core Ultra) as well as "Max Series" GPUs and Intel Gaudi accelerators. The platform supports software stacks including Habana and Intel oneAPI, among others. At the moment of writing this work, it does not provide access to FPGAs.

**Table 2.3:** Hardware availability across the main cloud platforms. Only reservable hardware is considered.

| Platform | CPU | GPU | AI accel. | FPGA | Manufacturers |
|---|---|---|---|---|---|
| AWS | ✓ | ✓ | ✓ | ✓ | AMD, AWS, Intel, NVIDIA, Qualcomm |
| Microsoft Azure | ✓ | ✓ | ✗ | ✓ | AMD, Intel, Microsoft, NVIDIA, |
| GCP | ✓ | ✓ | ✓ | ✗ | AMD, Google, Intel, NVIDIA |
| Alibaba Cloud | ✓ | ✓ | ✓ | ✓ | AMD, Intel, NVIDIA |
| Intel Tiber AI | ✓ | ✓ | ✓ | ✗ | Intel |

Table 2.3 summarises the aforementioned characteristics of each cloud platform, highlighting the heterogeneity in their hardware offerings as well as the diversity of vendors. As observed, GPUs dominate across all platforms, alongside specialised AIs accelerators such as NPUs and TPUs. Among the evaluated providers, AWS appears to offer the widest variety of hardware types and vendors. Concerning FPGAs, despite their programming complexity and more niche usage as discussed in Section 2.1, they are nevertheless significantly represented in major clouds due to the potential of their reconfigurable architecture. Although these cloud environments commonly incorporate devices such as DPUs or SmartNICs to accelerate internal operations, such resources are rarely made available for direct user reservation. These restrictions are justified given the reconfigurable nature of these platforms, including FPGAs, as offering them as reservable instances entails stricter security requirements compared to fixed-architecture devices.

### 2.2.4 Future directions and emerging trends

This subsection outlines future trends in distributed computing systems, based on the analysis presented in this section and the previous one, which examined heterogeneous computing.

On the one hand, modern HPC-oriented systems have seen a steady increase in **hardware heterogeneity**, aimed at addressing the specific computational demands of contemporary applications. This trend was clearly evidenced in Section 2.2.3, through the analysis of the Top500 evolution and the hardware offerings of major cloud providers. As repeatedly emphasised in Section 2.1, the growing diversity of accelerators across computing environments necessitates greater **standardisation** and improved tool support to enable efficient and portable execution.

On the other hand, the massive computational capacity of contemporary HPC systems, coupled with increasing environmental concerns, highlights the need to prioritise **energy-efficient computing**. As a result, the field of green computing is gaining significant importance and directly influencing the design of distributed systems. The Green500 [14] list demonstrates that it is indeed possible to balance high performance with energy efficiency. Future architectures are expected to further prioritise the minimisation of carbon footprints across the entire computing lifecycle, from hardware selection to workload scheduling.

Another clear trend is the shift towards **decentralised**, **edge-oriented systems** in response to the demands of emerging applications. Paradigms such as fog and edge computing address the large volumes of data generated by domains such as the IoT, enabling low-latency and real-time processing. However, their value extends beyond performance, as they offer advantages in terms of privacy, context awareness, and data localisation. Consequently, hybrid models that combine edge processing with the offloading of intensive tasks to centralised HPC infrastructures, such as clusters or cloud platforms, are gaining adoption.

In addition, it is interesting to analyse the role of modern FPGAs as accelerators in the current distributed systems. From a traditional compute acceleration perspective, these platforms offer considerable potential in tackling irregular workloads, as their reconfigurability enables the development of ad-hoc solutions that may outperform fixed architectures. Nonetheless, the lack of standard programming models and the inherent design complexity continue to limit their widespread adoption. Beyond compute acceleration, FPGAs are capable of processing data directly from the network, which has led to their use within NICs, forming SmartNICs. In distributed systems, where complex and high-bandwidth interconnects are common, this capability offers significant benefits in terms of latency and throughput. Moreover, FPGAs often surpass CPUs and GPUs in energy efficiency, owing to their fine-grained resource utilisation. These properties also propelled their use in edge applications: first, their ability to process data at the network level aligns well with edge principles that aim to bring computation closer to the data source; second, their low power consumption suits resource-constrained deployments. Therefore, FPGAs are key candidates for next-generation HPC systems that aim to combine sustainability with decentralised, edge-oriented design.

Although this analysis is shaped by the context of the present work, the trends identified above are consistent with findings from other studies. In particular, the

survey presented by Lindsay et al. [190] highlights key future directions, such as the specialisation and generalisation of accelerators to address hardware diversity, the growing importance of energy efficiency, and the rise of decentralised and edge-oriented architectures.

Chapter 3

# Evaluating SYCL for heterogeneous computing

*Finding a suitable architecture is not a simple task,*
*and finding an optimal one is even more challenging.*
— M. Fingeroff and T. Bollaert, *HLS Blue Book, 2010*

This chapter presents an analysis of the capabilities and performance offered by the SYCL programming model for heterogeneous computing. Although there are various implementations, as it was introduced in Chapter 2, this study is centred on the proposal of Intel, known as Intel oneAPI. To this end, a microbenchmark was developed to address two well-known iterative problems with different computational characteristics: memory-bound heat diffusion and compute-intensive image denoising. In both cases, dynamic workload balancing was applied between the host and the device. The heterogeneous configurations evaluated were CPU+GPU, comprising both integrated and discrete devices, and CPU+FPGA. In addition, the experiments were conducted in different environments, including a cloud platform, a supercomputer, and a high-end desktop machine. This study offers valuable insights into the behaviour of SYCL across different architectures, focusing on real iterative problems in which the workload is dynamically rebalanced between devices at runtime.

## 3.1 Time-step problems as a case study

To analyse the performance implications of the target heterogeneous computing schemes, two diffusion problems have been selected as case studies. These problems

were chosen because they are well-known and widely used, thus serving as easily understandable examples and representative cases of real iterative applications. Furthermore, due to their inherently iterative nature, they are particularly suitable for partitioning and distribution across multiple computing units. This is evidenced by the work of Lukarski and Neytcheva [191], which studied the performance gains enabled by parallel and heterogeneous computing strategies when applied to iterative methods. Further contributions in this direction include the study by Venkatasubramanian et al. [192], which tackled the Poisson equation through a multi-CPU and multi-GPU implementation of the Jacobi method. Benner et al. [193, 194] focused on using hybrid CPU+GPU systems to accelerate Newton's iteration for the matrix sign function. Agulleiro et al. [195] introduced a hybrid approach combining multicore CPUs with NVIDIA GPUs to enhance the efficiency of iterative tomographic reconstruction. Moreover, Halbiniak et al. [196] presented an OpenCL CPU+GPU solution for solidification modelling.

Since heat diffusion and image denoising are the two case studies used in this work, it is important to mention prior research that utilised heterogeneous computing to improve performance in tackling these problems. Belhaus et al. [197] presented a study where they compare parallel heat equation implementations on CPUs, as well as on NVIDIA GPUs. Sánchez et al. [198] developed an algorithm for impulsive noise removal in images, targeting solutions for multi-CPU, multi-GPU, and hybrid systems by the use of OpenMP and CUDA. Additionally, Sarjanoja et al. [199] introduced an implementation of the Block-Matching and Three-Dimensional (BM3D) denoising algorithm for CPU and GPU platforms, utilising OpenCL and CUDA.

### 3.1.1 Heat diffusion

As a representative example of isotropic diffusion, the two-dimensional heat equation is considered, expressed as:

$$\frac{\partial u}{\partial t} - D\Delta u = 0. \tag{3.1}$$

The equation asserts that the rate of change of temperature, $u$, relative to time at a given point is proportionate to the Laplacian of the temperature at that point, being the sum of the second-order partial derivatives of the temperature with respect to both dimensions $x$ and $y$. The physical interpretation of that equation is to illustrate how the temperature at a given point evolves over time, driven by temperature differences with neighbouring points and modulated by the material's thermal properties in the region, as governed by the diffusion coefficient $D$.

This problem is considered an isotropic method since it presumes that heat diffuses in a uniform way along all directions, regardless of the orientation of the coordinate

axes, as stated in Equation (3.1). The reason for this is the scalar differential Laplacian operator, invariant to rotations of the coordinate axes.

The initial conditions employed for solving the problem are

$$u(x, y, 0) = \sin(x)\sin(y), \tag{3.2}$$

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

Equation (3.2) represents a suitable choice for the initial temperature distribution, as it adheres to two key conditions. Firstly, it is an inherently smooth function, devoid of discontinuities, sharp transitions, or singularities, thereby ensuring that the temperature remains continuously differentiable throughout the domain. Secondly, it naturally satisfies that the temperature at the boundary is zero, as illustrated in Figure 3.1, since

$$u(0, y, t) = u(x, 0, t) = u(\pi, y, t) = u(x, \pi, t) = 0. \tag{3.3}$$

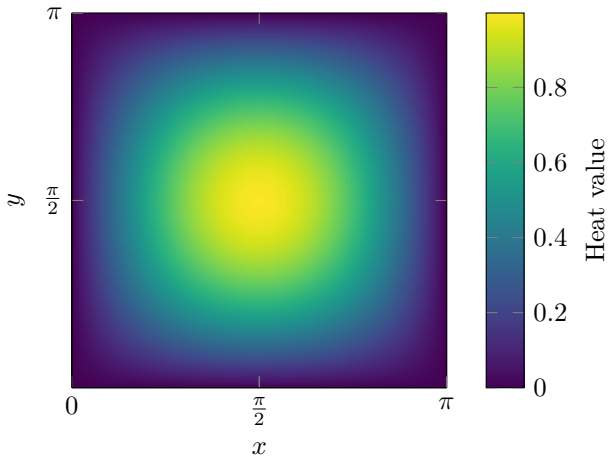



**Figure 3.1:** Representation of the initial temperature distribution $u(x, y, 0)$.

Accordingly, the problem can be formally stated as:

$$\begin{cases} \frac{\partial u}{\partial t} - D\Delta u = 0, \\ u(x, y, 0) = \sin(x)\sin(y), \\ u(0, y, t) = u(x, 0, t) = u(\pi, y, t) = u(x, \pi, t) = 0. \end{cases} \tag{3.4}$$

#### 3.1.1.1 Discretisation

To numerically solve the heat equation using the finite difference method, both the spatial domain and the temporal interval must be discretised. For the spatial domain $\Omega = (0, \pi) \times (0, \pi)$, it can be discretised using a structured mesh composed of $N \times N$ evenly distributed nodes. The spacing between adjacent nodes in the $x$ and $y$ directions is denoted by $\delta_x$ and $\delta_y$, respectively, with $\delta_x = \delta_y = \frac{\pi}{N}$.

For the temporal discretisation, a constant time step $\delta_t$ is adopted. The time integration is carried out using the Explicit Euler scheme, defined as

$$u(x, y, t + \delta_t) = u(x, y, t) + \delta_t \frac{\partial u(x, y, t)}{\partial t}, \tag{3.5}$$

where $u(x, y, t)$ denotes the temperature at a position $(x, y)$ and time $t$.

To approximate the spatial derivatives, the well-known five-point stencil [200] is used, shown in Figure 3.2. The computational domain is partitioned into a regular grid with uniform spacing in both $x$ and $y$ directions, where $\delta_x$ and $\delta_y$ represent the distances between neighbouring grid points along the $x$ and $y$ axes, respectively.

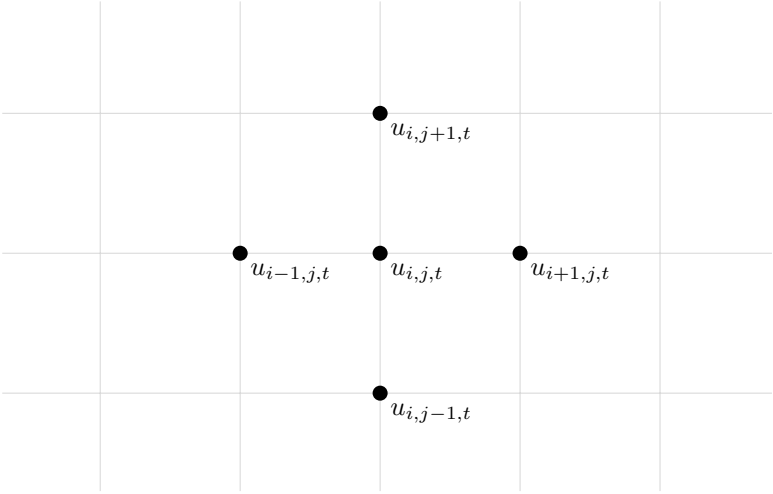



**Figure 3.2:** Finite difference five-point stencil for approximating the Laplacian at grid point $(i, j)$ and time $t$. The central value $u_{i,j,t}$ depends on its four immediate neighbours.

The second-order partial derivatives with respect to $x$ and $y$ can be approximated using central finite differences as

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1,j,t} - 2u_{i,j,t} + u_{i-1,j,t}}{\delta_x^2}, \quad \frac{\partial^2 u}{\partial y^2} \approx \frac{u_{i,j+1,t} - 2u_{i,j,t} + u_{i,j-1,t}}{\delta_y^2}. \tag{3.6}$$

By substituting these finite difference approximations into the original heat equation, the following discrete formulation is obtained:

$$u_{i,j,t+\delta_t} = u_{i,j,t} + \delta_t \left( \frac{u_{i+1,j,t} - 2u_{i,j,t} + u_{i-1,j,t}}{\delta_x^2} + \frac{u_{i,j+1,t} - 2u_{i,j,t} + u_{i,j-1,t}}{\delta_y^2} \right), \tag{3.7}$$

which expresses the temperature at point $(i, j)$ at time $t + \delta_t$ in terms of the temperature at the same point and its four immediate neighbours at time $t$. This formulation allows the temperature at each grid point to be updated iteratively at each time step. This procedure corresponds to an explicit time integration method and requires that the time step $\delta_t$ be sufficiently small to maintain numerical stability.

Assuming a square computational domain and uniform grid spacing in both directions, i.e. $\delta_x = \delta_y = \delta$, the computations would be:

$$u_{i,j,t+\delta_t} = u_{i,j,t} + \delta_t \left( \frac{u_{i+1,j,t} + u_{i,j+1,t} - 4u_{i,j,t} + u_{i-1,j,t} + u_{i,j-1,t}}{\delta^2} \right). \tag{3.8}$$

### 3.1.2 Image denoising

The second problem evaluated is image denoising, which lends itself well to diffusion-based techniques. In particular, it was considered the approach by Tang et al. [201], which proposed an anisotropic diffusion method to suppress noise in digital images. For simplicity, the implementation used in this work concentrates solely on the anisotropic brightness flow component introduced in their method. Hence, any grey-scale image with the spatial dimensions indicated in Section 3.4 can be employed to replicate this study.

#### 3.1.2.1 Anisotropic diffusion for brightness

A digital image can be understood as a discrete two-dimensional representation of a scene, structured as a finite grid of pixels. Each pixel corresponds to a specific spatial location and carries a value that encodes either the intensity or colour of the light captured at that point. In grey-scale images, this value consists of a single channel that represents the brightness or intensity at each pixel. Conversely, colour images are typically composed of three separate channels, each representing the intensity of the red, green, and blue (RGB) components at every pixel location.

Therefore, a digital image may be represented as a function $f : \mathbb{R}^2 \to \mathbb{R}^c$ where each coordinate pair $(x, y)$ is associated with a $c$-dimensional vector of real numbers. This vector encodes the intensity or colour information at each corresponding point. In practice, the image is often treated as a matrix $M(x, y)$, in which each entry contains the relevant pixel values across the $c$ channels.

In the context of diffusion-based image processing, Tang et al. [201] proposed two distinct methods for handling grey-scale images, or more specifically, brightness information. The first approach relies on isotropic flow, formulated through the classical Laplace equation:

$$\begin{aligned} &\frac{\partial M}{\partial t}(x,y,t) = M_{xx}(x,y,t) + M_{yy}(x,y,t) = \Delta M(x,y,t), \\ \text{where} \quad &M_{xx}(x,y,t) = \frac{\partial^2 M}{\partial x^2}(x,y,t), \quad \text{and} \quad M_{yy}(x,y,t) = \frac{\partial^2 M}{\partial y^2}(x,y,t). \end{aligned} \tag{3.9}$$

The second approach, which is the diffusion model adopted in this work, corresponds to the anisotropic flow. This is captured by the following equation, where the term $1/\left(1 + ||\nabla M||\right)$ introduces the anisotropic behaviour:

$$\begin{aligned} &\frac{\partial M}{\partial t}(x,y,t) = \frac{\left(M_{xx}M_y^2 - 2M_xM_yM_{xy} + M_{yy}M_x^2\right)^{\frac{1}{3}}}{1 + \|\nabla M\|}, \\ \text{where} \quad &M_x = \frac{\partial M}{\partial x}, \quad M_y = \frac{\partial M}{\partial y}, \quad M_{xx} = \frac{\partial^2 M}{\partial x^2}, \\ &M_{yy} = \frac{\partial^2 M}{\partial y^2}, \quad \text{and} \quad M_{xy} = \frac{\partial^2 M}{\partial x \partial y}. \end{aligned} \tag{3.10}$$

It is worth noting that solving Equation (3.10) entails a significantly higher arithmetic workload than the simpler isotropic model in Equation (3.9), which reproduces the scenario discussed in Section 3.1.1. As a result, these two models exhibit differing levels of arithmetic intensity, allowing for a meaningful comparison of how each computational workload interacts with the architectural features of the hardware under study.

#### 3.1.2.2 Discretisation

In image processing, spatial discretisation is typically straightforward, as the image resolution inherently defines a discrete grid of pixels:

$$\delta_x = \delta_y = 1. \tag{3.11}$$

This implies that each spatial location within the image corresponds to a discrete value; thus, the problem is reduced to identifying suitable mathematical operations to apply to these values.

The implementation adopted in this study uses the Explicit Euler method with a constant time step for temporal discretisation. This choice facilitates the maintenance of both numerical stability and solution accuracy throughout the simulation.

## 3.2 Intel oneAPI

Intel oneAPI [17], introduced in Chapter 2, is a unified software suite designed to facilitate application development across various hardware architectures. It offers a collection of compilers, libraries, and development tools that enable high-level software to execute on CPUs, GPUs, FPGAs, and other accelerators. Furthermore, Intel oneAPI supports multiple programming languages, including C++, Fortran, and DPC++ [18]. The latter is based on C++17 and features a modern, heterogeneous programming model built upon the SYCL standard. The toolkit also provides a robust set of utilities for profiling, debugging, and optimising code for different architectures, such as Intel Advisor [202] or Intel VTune [203].

Regarding memory management, Intel oneAPI offers two primary approaches: USM and buffers with accessors. The USM model adopts a pointer-based methodology, featuring syntax familiar to C++ programmers, enabling memory allocation on the host (`malloc_host`), on the device (`malloc_device`), or shared between them (`malloc_shared`). Shared allocations are particularly advantageous when the host and the device frequently access the same data, as it remains accessible to both parties. In this case, the data transfers are managed transparently by runtime components and low-level drivers. For device-specific allocations, memory resides in the device's local memory, and data transfers must be handled explicitly by the developer through copy operations between the host and the device. In addition to USM, buffers provide an abstraction of memory regions on the device, while accessors serve as the mechanism to request read or write access to these buffers. According to the SYCL 2020 specification [101], the SYCL runtime is responsible for managing the allocation and deallocation of buffers.

Referring to kernel implementation, Intel oneAPI offers different constructs that can be used on various devices. The way to interact with any platform is through a queue, as is standard for tools that present a unified programming model. The common options for developing kernels are `parallel_for`, which includes the possibility of using `nd_range`, and `single_task`. For GPUs, `parallel_for` and `nd_range` are commonly used, since they express data-parallel workloads that are well suited to their architecture [204]. The main difference between these two mechanisms is that `nd_range` provides more control over thread and work-group layout, which can be useful for tuning GPU performance. In contrast, the general recommendation for FPGAs is to use `single_task` kernels, as they enable sequential execution and deep pipelining, allowing the compiler to generate highly optimised custom logic [205].

The implementation of offloaded kernels is closely tied to the compilation model, as the chosen approach directly influences the way the device code is generated, packaged, and executed. In particular, Intel oneAPI supports both Just-in-Time (JIT) and Ahead-of-Time (AOT) flows. In the JIT flow, device code is compiled into Standard Portable Intermediate Representation (SPIR)-V, designed for portable parallel com-

pute and graphics programming. It is then embedded alongside the host executable into a fat binary, a single executable file that contains code for multiple architectures or targets. This approach is not supported for FPGAs, which is understandable given the necessity to resolve design-specific details at compile time to synthesise the reconfigurable hardware. At runtime, the SPIR-V is translated into device-specific binary code depending on the available hardware, offering flexibility at the cost of increasing runtime overhead. In contrast, the AOT flow performs a complete device code generation at compile time for a specified target, embedding the result directly into the fat binary. Although this reduces deployment flexibility, it leads to lower runtime latency and facilitates more efficient debugging. In both cases, the fat binary enables unified execution on the host and the device by encapsulating all necessary code in a single, portable executable. In addition, Intel oneAPI supports several FPGA-specific compilation modes designed to facilitate iterative development, tailored to the unique requirements of reconfigurable hardware. These modes include emulation, simulation, optimisation report generation, and hardware image compilation. During the emulation, the device code is compiled to be run on a CPU for functional validation. In the simulation stage, cycle-accurate debugging is enabled via Questa* [206], the simulation environment provided by Intel. Optimisation report generation produces intermediate compilation artefacts and resource utilisation estimates to guide design refinement. Finally, the hardware image compilation generates either a bitstream for deployment on acceleration boards or RTL files for integration into larger designs using Intel Quartus Prime [19]. A typical development workflow involves iterating through emulation, simulation, and optimisation report stages prior to the generation of the final hardware image.

### 3.2.1 Intel oneAPI in heterogeneous computing schemes

Since its market introduction in 2020, Intel oneAPI has been adopted to develop heterogeneous computing solutions in diverse fields. For instance, Constantinescu et al. [207] implemented Markov decision processes on low-power CPU+GPU SoCs, comparing OpenCL + TBB against oneAPI + TBB in terms of programmability and performance. Yong et al. [208] migrated an ultrasound imaging application from CUDA to oneAPI, optimising it for GPU, FPGA, and CPU platforms. Lupescu and Ţăpuş [209] developed a hybrid hashtable targeting a SoC composed of a CPU and an Integrated GPU (iGPU). Marinelli et Raja [210] ported a GPU-based hash join algorithm from CUDA to DPC++ and evaluated it across Discrete GPUs (dGPUs), iGPUs, and multicore CPUs. They further introduced OneJoin [211], an edit similarity join between architectures for DNA data storage. Nozal and Bosque [212] assessed the performance and energy efficiency of prominent HPC benchmarks using dynamic and static workloads on CPU+GPU heterogeneous systems. Bavarsad et al. [213] presented HosNa, a benchmark suite implemented in DPC++ for heterogeneous architectures, alongside a performance analysis of Intel-manufactured CPUs and FPGAs

using algorithms from the suite. Kashino et al. [214] explored the use of GPU+FPGA for complex physical simulations. Groth et al. [215] proposed a hybrid accelerated implementation of various hash table operations for dGPUs and Accelerated Processing Units (APUs), the latter being a combination of a CPU and an iGPU. In particular, they used Intel oneAPI to develop the search operation for the APU. Li et al. [216] developed a portable framework for large-scale graph processing leveraging a heterogeneous CPU+GPU architecture. More recently, Balaji et al. [217] benchmarked complex multiplication in signal processing across CPU, GPU, and FPGA platforms using a unified Intel oneAPI codebase, offering valuable guidance for selecting optimal hardware in real-time signal processing contexts. Liang et al. [218] demonstrated the feasibility of true multi-hybrid acceleration by coupling NVIDIA GPU and Intel FPGA devices within a single oneAPI-based environment, expanding the scope of heterogeneous computing to different-vendor architectures. Finally, Campos et al. [219] explored data flow design and vectorisation strategies for streaming applications on heterogeneous CPU+GPU platforms.

## 3.3 Implementation details

To conduct this study, a microbenchmark named `ihp-sycl`[1] was developed. This section outlines the specific implementation details concerning kernel design, memory management, and dynamic workload balancing.

### 3.3.1 Kernels design and memory management

In this study, oneAPI's mechanisms are employed to manage memory, coordinate device execution, and express parallelism. For the latter, the `parallel_for` member function of the SYCL handler class [104, 204] is utilised to define and launch the kernels. It offers an interface to execute a SYCL kernel in parallel over a defined range of values. The implementation adopts the two-parameter version, in which the first parameter specifies the range and the second is a lambda function containing the kernel code to be executed on the device. In this case, workloads are distributed in a row-wise basis, since the input data is either a matrix or an image, structures which are conceptually equivalent. Therefore, the range is expressed with `sycl::range(size, cols)` where `size` represents the number of rows assigned to the device for a given iteration set, and `cols` denotes the total number of columns in the input matrix or image. A detailed description of the workload distribution across devices will be provided in Section 3.3.2. On the host side, the code is parallelised using the OpenMP library and, in particular, the `omp parallel for` directive.

[1] https://github.com/silviaralcaraz/ihp-sycl

Although it is generally expected that achieving maximum performance may require device-specific implementations, the same kernel is used on all the devices evaluated, as this work examines whether functional kernels can be generated for multiple device types without modifying the source code by leveraging oneAPI's portability. The only adjustments made to the GPU kernel for FPGA deployment were minor and stylistic, introduced to meet the specific design requirements of these devices, such as structural and memory allocation constraints.

For GPU execution, the developed microbenchmark supports multiple compilation modes, including configurations compatible with NVIDIA devices through the Codeplay plugin [119]. In such cases, the `-fsycl-targets=nvptx64-nvidia-cuda` flag is included during compilation, and the appropriate target device is selected at runtime using the `ONEAPI_DEVICE_SELECTOR=cuda:gpu` environment variable. This flexibility enables the same source code to be compiled and executed across GPUs with different capabilities and vendors.

Finally, AOT compilation is employed as the target hardware is known at compile time for all experiments, and it is the only supported mode for FPGAs, as previously mentioned. Furthermore, utilising AOT compilation consistently across devices enhances execution efficiency by enabling more extensive optimisations during compilation and simplifies the build process for heterogeneous targets.

### 3.3.2 Dynamic workload balancing

A key consideration in heterogeneous computing is the effective distribution of workload among the target devices. The aim is to exploit the distinct architectural strengths of each one to maximise overall performance.

To address this, a dynamic load balancing strategy has been adopted, motivated by two main factors. On the one hand, employing a static approach would necessitate either an initial profiling phase, incurring additional overhead, or a manual-tuning process to determine the most efficient workload partitioning for each device pairing. On the other hand, it is possible for one of the devices to suffer occasional performance drops due to system-level routines or other external factors, which would deviate from an optimal balance.

Thus, as this work deals with diffusion problems that are inherently iterative, dynamic workload balancing has been implemented using Iterative Heterogeneous Parallelism (IHP) [1]. It is a dynamic parallel programming scheme specifically designed for iterative or time-stepped methods on heterogeneous platforms. At runtime, the workload is distributed among devices based on their current performance, with the goal of minimising overall execution time. The library also allows users to select the type of work partitioning, which may be advantageous depending on the particular structure of the input data. As described in Section 3.3.1, this work considers a row-wise distribution model, in which each row is treated as the fundamental task unit.

Within IHP library, the parameter $\alpha \in [0, 1]$ denotes the amount of work assigned to the CPU. In the extrema, $\alpha = 1$ implies that the CPU handles the whole computation, while $\alpha = 0$ means that all work is completely offloaded to the accelerator. Intermediate values indicate that the workload is distributed between the host and the device in ratios $\alpha$ and $1 - \alpha$, respectively.

## 3.4 Experimental setup

The experiments were conducted across three different computational environments: two distributed computing systems, a cloud platform and a supercomputer, and a high-end desktop machine. The cloud system was the unfortunately now-deprecated Intel DevCloud[2], which provided access to a range of Intel-manufactured CPUs, GPUs, and FPGAs. This environment was chosen for performing the experiments with the CPU+iGPU and CPU+FPGA configurations. Intel dGPUs were not evaluated, as only iGPUs were available in the cloud at the time this study was carried out. The version of Intel oneAPI employed in these experiments was 2023.1.0, which was the version available on the cloud during development.

For the evaluation of dGPUs and their combined configurations, two systems were utilised: a supercomputer, the FT3, hosted by the CESGA, and a headless desktop machine. Two different NVIDIA dGPU models were used, including an HPC dGPU as well as a high-end desktop dGPU. This approach enables a more comprehensive analysis and supports richer conclusions due to the distinct capacities of the devices. In these experiments, the microbenchmark was updated for compatibility with Intel oneAPI version 2025.1.0, using the corresponding version of the Codeplay plugin to target NVIDIA GPUs, as detailed in Section 3.3.1.

The specific hardware devices associated with each evaluated configuration are listed in Table 3.1. Additionally, a full description of each node or machine configuration, along with the hardware capabilities of the devices, is provided in Appendix A.

**Table 3.1:** Hardware configurations used in the experiments.

| Scheme | Host | Device |
|---|---|---|
| CPU+iGPU | Intel Xeon E-2176G | Intel UHD Graphics P630 |
| CPU+dGPU (HPC) | Intel Xeon Platinum 8352Y | NVIDIA A100 |
| CPU+dGPU (desktop) | Intel Core i9-12900K | NVIDIA RTX 3080 Ti |
| CPU+FPGA | Intel Xeon Gold 6128 | Intel Arria 10 |

In order to ensure the reproducibility of the results, the microbenchmark's parameters chosen for running the experiments are explicitly detailed. These include, among

[2]No longer available; it has been restructured and renamed as Intel Tiber AI Cloud [189].

others, the matrix dimensions evaluated, the initial CPU workload ratio—denoted as $\alpha$ throughout this work—, the data type used, and the memory allocations employed.

The experiments were conducted using square matrices of size $N \times N$, where $N$ takes the following values: 512, 800, 1024, 1500, 2048, 3000, 4096, 6000, 8192, 10 000, 16 384, and 20 000. For experiments involving the FPGA, the largest matrix size was limited to 10 000, due to the comparatively higher execution times observed on this platform. Even so, this size was deemed sufficient to assess the performance of the proposed approach. These matrix sizes were selected to cover a broad spectrum, ranging from small to large problem dimensions, and include both powers-of-two and non-power-of-two sizes. This selection enables the analysis of scalability while also allowing the investigation of whether matrix sizes of the form $2^n$ are handled differently by the hardware or software, which may lead to performance anomalies.

With respect to numerical precision, all computations were carried out using 32-bit single-precision floating-point arithmetic. This choice represents the most balanced trade-off between numerical fidelity and computational efficiency across the three architectures under investigation. Generally, GPUs achieve their highest arithmetic throughput and memory bandwidth utilisation in single precision, particularly in iGPUs and mid-range dGPUs. CPUs also benefit from the halved memory footprint, which alleviates cache pressure. In the case of FPGAs, their reconfigurable architecture can be adapted to different data widths through dedicated kernels, but smaller word lengths contribute to decreasing resource utilisation. That said, due to the problem's nature and the kernel's design, specifically its exploitation of data parallelism, GPUs gain a distinct advantage in this scenario, as their architecture is particularly well-suited to efficiently handle parallel workloads. Moreover, the two benchmark problems examined tolerate the precision offered by the `float` data type without compromising correctness. Therefore, doubling the mantissa by using `double` precision yields negligible benefit in terms of solution quality.

Regarding the CPU-device workload distribution, $\alpha_0$ was tested with values of 0, 0.5, and 1. A value of $\alpha_0 = 0$ corresponds to the case in which the device (e. g., GPU or FPGA) performs the whole computation; $\alpha_0 = 1$ denotes full CPU execution; and $\alpha_0 = 0.5$ represents an initial balanced distribution of workload between the CPU and the device. In the latter case, $\alpha$ changes dynamically throughout the iterations. The choice of $\alpha_0 = 0.5$ for the heterogeneous scenario assumes an ideal condition where both computing resources have equal capabilities. While this assumption rarely reflects practical systems, it serves as a neutral starting point, relying on dynamic load-balancing mechanisms to adjust the ratio during execution.

Concerning memory management, device USM allocations were utilised, as they provide more control over memory handling and the way data transfers are performed. Additionally, all the experiments were executed over 1000 iterations to improve statistical robustness and result reliability. Both use cases, heat diffusion and image denoising, were evaluated under identical experimental conditions.

Finally, since the host-side code was parallelised using OpenMP, it is impor-

tant to mention that all the experiments were conducted using the default value of `OMP_NUM_THREADS`, which corresponds to the maximum number of cores, both physical and logical, available on each CPU.

## 3.5 Results and discussion

This section presents and discusses the results from the experiments described in Section 3.4. It also provides a final comparison of execution times and floating-point computational throughput across all the devices and computational schemes evaluated in this work, for the problems considered as case studies.

### 3.5.1 CPU+iGPU approach

This subsection includes the results obtained using the Intel Xeon E-2176G [220] and its integrated GPU, the Intel UHD Graphics P630, within the Intel DevCloud environment for the two case studies considered. The complete hardware configuration of the iGPU node is described in Appendix A.1.

For the heat diffusion problem, the results reveal that the CPU parallelised with OpenMP achieves significantly better execution times than the iGPU across nearly all the tested matrix sizes. This is depicted in Figure 3.3(a), where the execution times have been normalised with respect to the CPU times. In configurations utilising both devices, some performance improvements over the faster device are observed for certain problem sizes. Nevertheless, in most cases, the CPU-only execution outperforms the hybrid approach, which is not the expected behaviour. Concerning the workload distribution parameter, $\alpha$, it is aligned with the execution times observed under the CPU+iGPU configuration. Initially, higher $\alpha$ values assign a greater computational load to the CPU, which is reasonable given the relatively poor performance of the iGPU for these matrix sizes. Subsequently, $\alpha$ stabilises near 0.5, providing a roughly balanced workload distribution between the two devices. This implies that, under these conditions, the CPU and the iGPU deliver comparable performance when working together.

To further understand the observed outcomes, additional experiments were conducted, including an analysis of the time each device required to process a single row and the evolution of $\alpha$ over successive iterations. For this purpose, the size $N = 6000$ was selected as it was anticipated that the GPU performance would improve at this scale, based on the trend illustrated in Figure 3.3(a). The corresponding results are presented in Figure 3.4(a). Beginning with the progression of $\alpha$, the initial values allocated the majority of the workload to the CPU, which is consistent with the relatively high execution times observed for the iGPU during the first $\approx$10 iterations.

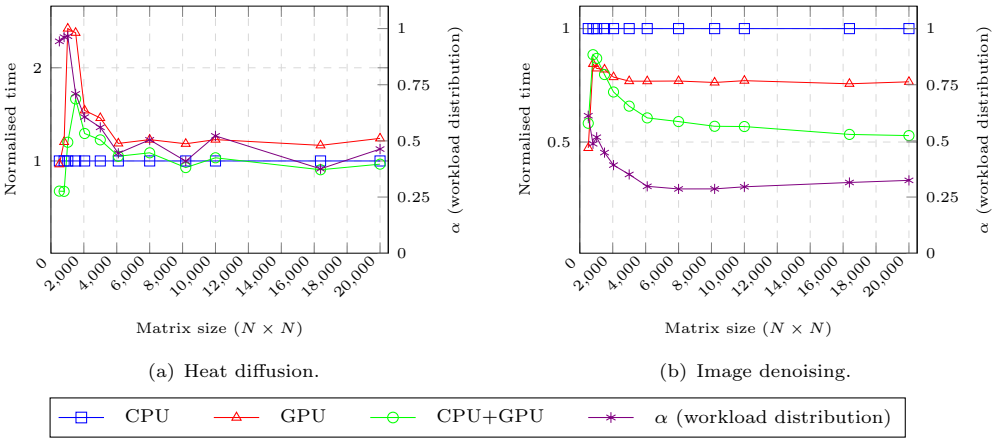



**Figure 3.3:** Normalised execution times relative to the CPU (left axis; lower is better) and workload distribution $\alpha$ (right axis; $\alpha = 1$ means CPU-only, $\alpha = 0$ means device-only) across different matrix sizes for the CPU+iGPU scheme.

Then, $\alpha$ stabilised around 0.5, resulting in an approximately balanced distribution of computation between the two devices. This behaviour was expected, as both of them demonstrated similar performance during those iterations. Examining the results obtained from executing the experiments on the CPU and the iGPU separately, the CPU consistently outperformed the iGPU, albeit slightly. Nonetheless, the collaboration between the two devices led to a degradation in performance, resulting in similar outcomes from each, as anticipated earlier.

To clarify this behaviour in more detail, experiments were performed in which the algebraic computational workload was increased by several orders of magnitude while maintaining the same memory accesses. Under these conditions, the iGPU demonstrated enhanced performance relative to the CPU by leveraging its architecture advantages. Furthermore, the overhead arising from contention between the CPU and the iGPU for memory access was diminished. Consequently, the results for the hybrid configuration exhibited improved execution times, surpassing those of the fastest device. Considering these factors, the initially commented results can be attributed to the device's limited memory bandwidth, which degrades its performance in an inherently memory-bound problem. The CPU+iGPU scheme does not yield improved execution times, as both share the same communication bus and are limited by memory bandwidth.

As discussed in Section 3.1, the image denoising method presents a more computationally demanding problem than the heat diffusion, enabling the iGPU to exploit its high degree of parallelism. Despite being an integrated device, the iGPU outperforms the CPU, by up to 25 % within the context of this problem. Moreover, as is expected

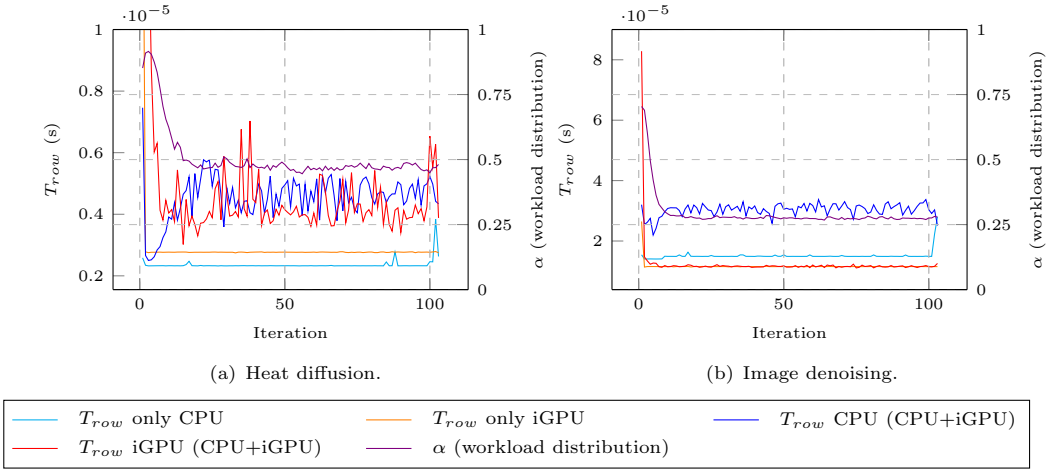



**Figure 3.4:** Evolution of time to compute a row ($T_{row}$) and $\alpha$ using CPU-only, iGPU-only, and CPU+iGPU configurations for (a) heat diffusion and (b) image denoising problems.

in a theoretical scenario, the combination of both devices working together provides results that surpass the fastest device, as each contributes to the overall computation. Concerning $\alpha$, in this case, its value generally remains below 0.5, indicating that the dynamic workload distribution is giving a greater share of tasks to the iGPU. This allocation is consistent with the superior performance demonstrated by the iGPU, as illustrated in Figure 3.3(b).

In conclusion, this case study demonstrates that heterogeneous computing schemes offer clear performance benefits for problems where the architectural strengths of each device can be effectively exploited. In contrast to the heat diffusion scenario, Figure 3.4(b), generated with identical parameters as in the previous case, illustrates that the iGPU exhibits a largely consistent behaviour when processing rows over successive iterations, both when operating independently and in collaboration with the CPU. Additionally, the CPU performance is somewhat reduced in the hybrid configuration due to the additional overhead incurred by managing communications with the iGPU. This fact is reflected in both Figures 3.4(a) and 3.4(b), where the CPU takes on average approximately twice as long per row in the heterogeneous scheme compared to the CPU-only execution. In any case, in the image denoising scenario, the performance of the iGPU compensates for this increase in the host processing time. Regarding $\alpha$, it remains relatively stable, allocating the majority of the workload to the iGPU. This is explained by the consistent row-processing time of the iGPU throughout the iterations, which enables the dynamic load balancing mechanism to operate efficiently with minimal variation across iterations.

### 3.5.2 CPU+dGPU (HPC) approach

This subsection comprises the results obtained with the Intel Xeon Platinum 8352y [221] and the NVIDIA A100 [222] dGPU, within the FT3 supercomputer for the two case studies considered. The complete hardware configuration of the NVIDIA A100 node is described in Appendix A.3.

With this HPC configuration, the value of $\alpha$ tends towards zero, indicating that nearly all the computational work is offloaded to the device in both case studies, as shown in Figures 3.5(a) and 3.5(b). This behaviour is expected, given the substantial computational capacity and high memory bandwidth of the dGPU. Hence, the contribution of the CPU within the heterogeneous scheme becomes almost negligible and does not compensate for the overheads associated with communication between the host and the device during the workload redistribution. These limitations are particularly evident in scenarios involving small matrices, where the underutilisation of the dGPU amplifies the relative impact of the communication overheads. This issue is clearly depicted in Figure 3.5(a), related to the memory-bound heat diffusion problem, where performance improves significantly only when $N > 2000$, as $\alpha$ assigns more load to the device. In contrast, the image denoising problem, shown in Figure 3.5(b), demonstrates that load balancing between the host and the device offers no clear benefit. This is due to the high degree of parallelism and memory bandwidth of the HPC dGPU, which allows it to handle the workload independently and efficiently.

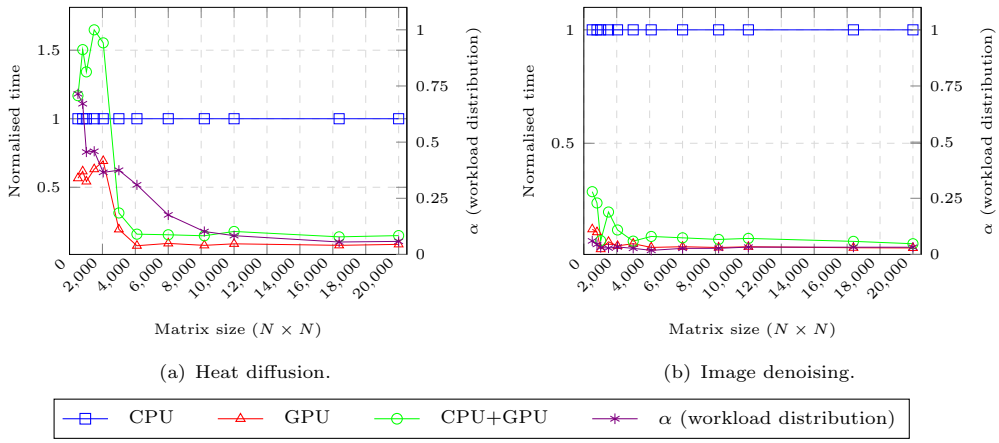



**Figure 3.5:** Normalised execution times relative to the CPU (left axis; lower is better) and workload distribution $\alpha$ (right axis; $\alpha = 1$ means CPU-only, $\alpha = 0$ means device-only) across different matrix sizes for the CPU+dGPU (HPC) scheme.

### 3.5.3 CPU+dGPU (desktop) approach

This subsection summarises the results obtained with the Intel Core i9-12900K [223] and the NVIDIA RTX 3080 Ti [224] dGPU, on a headless desktop machine for the two case studies considered. The complete hardware configuration of the system is described in Appendix A.5.

As depicted in Figures 3.6(a) and 3.6(b), the dGPU consistently outperforms the CPU across both benchmarks. However, the performance gap is less pronounced than in the HPC configuration with the NVIDIA A100, particularly in the image denoising case. For this reason, the workload sharing between the host and the device yields a modest gain over the dGPU-only implementation. This improvement, approximately 7%, is observed only in the more computationally demanding problem involving large matrices, specifically from $6000 \times 6000$ onwards.

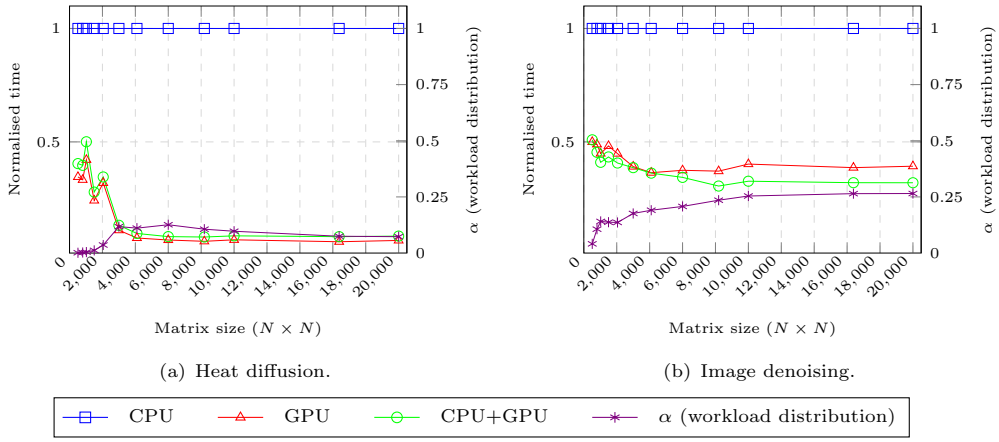



**Figure 3.6:** Normalised execution times relative to the CPU (left axis; lower is better) and workload distribution $\alpha$ (right axis; $\alpha = 1$ means CPU-only, $\alpha = 0$ means device-only) across different matrix sizes for the CPU+dGPU (desktop) scheme.

### 3.5.4 CPU+FPGA approach

This subsection includes the results obtained with the Intel Xeon Gold 6128 [225] and the Intel Arria 10 [226] FPGA, within the Intel DevCloud environment for the two case studies considered. The complete hardware configuration of the FPGA node is described in Appendix A.2.

Concerning the heat diffusion problem, the results demonstrate that employing a non-specific implementation for the FPGA kernel translates into non-competitive performance. Across all the problem sizes, the CPU offers substantially faster execution times than the FPGA. In the heterogeneous configuration involving both the

CPU and the FPGA, no improvement is observed over the CPU-only execution across any of the matrix sizes considered. The workload distribution parameter, $\alpha$, fluctuates consistently in response to the performance of each device. The FPGA exhibits marginally better performance at specific matrix sizes, and it is in these cases that the library adjusts $\alpha$ to allocate a portion of the workload to it. For other sizes where such distribution proves inefficient, $\alpha$ reverts to assigning almost the entire workload to the CPU. In this scenario, host-device communication overhead becomes a critical performance factor. It is noteworthy that, as it has been previously mentioned, the implementation uses a generic kernel intended for portability across multiple target devices. Thus, not using device-specific optimisations might result in a suboptimal design where these transfers are very costly. Experimental observations suggest that the resulting design generated by the HLS process can mitigate the cost of some transfers while exacerbating others. This behaviour manifests as the saw-tooth pattern in Figure 3.7(a). This phenomenon arises from the influence of word size on data movement in FPGAs. The width of the data word determines how efficiently memory can be accessed and transferred without the need for additional padding or alignment, which incurs overheads. Based on the limited design details accessible through the oneAPI reports, where low-level implementation aspects are largely abstracted, the memory interface to the Double Data Rate (DDR) appears to operate with a 512-bit data width. Apart from alignment-related constraints, performance drops when matrix dimensions are not powers of two. In such cases, the compiler increases memory port widths to the nearest greater power of two and masks out the extra bytes [227], resulting in unnecessary bandwidth consumption and degraded performance. Consequently, matrix sizes that are powers of two yield significantly better execution times due to more efficient memory transfers and reduced communication overhead across the iterations.

For the image denoising problem, the results exhibit similar trends to those observed previously, with the CPU again providing superior execution times. However, as depicted in Figure 3.7(b), two key differences influence the performance in this case. First, the disparity between the execution times of the CPU and the FPGA is less pronounced than that observed in the heat diffusion experiments. Second, although the FPGA still performs worse than the CPU, its behaviour across varying matrix sizes is more stable than in the previous scenario. This increased consistency facilitates a more effective application of the dynamic workload distribution between the CPU and the FPGA. These two factors explain the viability of the heterogeneous approach in outperforming the individual performance of each device. Nonetheless, such improvements are only feasible when the matrix dimensions are powers of two, as previously discussed. In other instances, memory transfer overheads dominate, making the communication costs prohibitive and undermining the benefits of load sharing between the host and the accelerator.

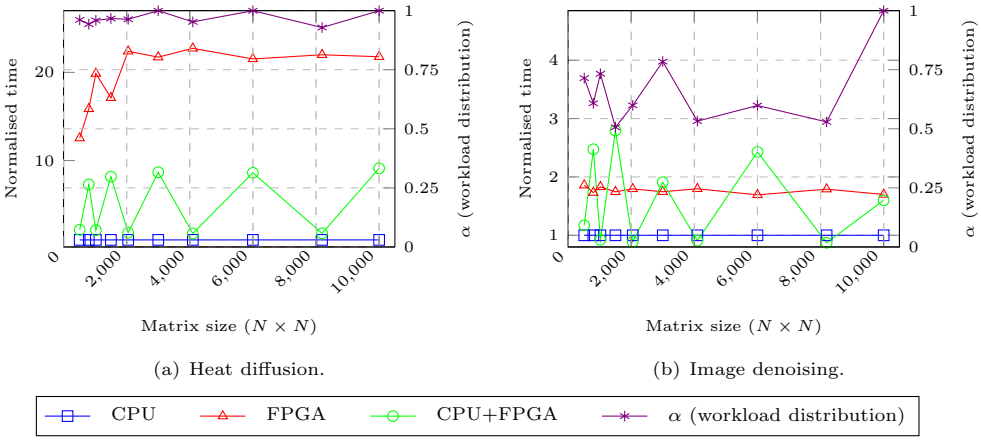



**Figure 3.7:** Normalised execution times relative to the CPU (left axis; lower is better) and workload distribution $\alpha$ (right axis; $\alpha = 1$ = CPU-only, $\alpha = 0$ = FPGA-only) across different matrix sizes for the CPU+FPGA scheme.

### 3.5.5 Performance across devices and heterogeneous schemes

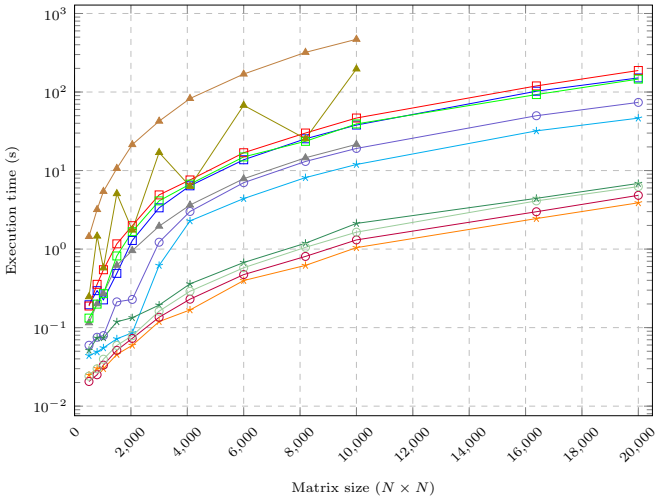
This subsection provides a comprehensive performance analysis of the heterogeneous computing configurations and the individual devices utilised in the experiments. Execution times and real floating-point computational throughput are compared across the hybrid schemes and the assessed CPUs, iGPU, dGPUs, and FPGA. To calculate the Giga FLOPS (GFLOPS), the kernels used were characterised according to their computational intensity.

Before presenting the specific results, it is worth noting that the host device varies among the heterogeneous schemes, as detailed in Section 3.4. This variation affects the overall performance of the hybrid schemes due to the different capabilities of each CPU, and should be taken into account.

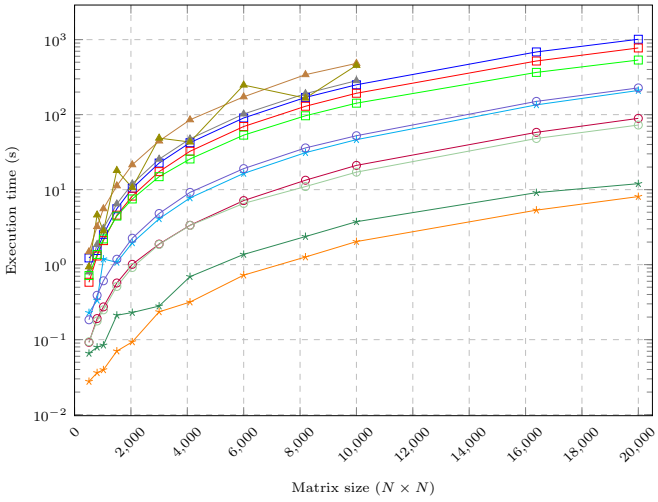
As illustrated in the results shown in Figure 3.8(a), for the heat diffusion problem, the best performance is attained by the NVIDIA A100, closely followed by the other dGPU model, the RTX 3080 Ti. This advantage becomes more pronounced as the problem size increases, a trend more clearly observed in Figure 3.9, where, from $N = 1024$ onwards, the A100 achieves higher GFLOPS. Specifically, in that case, the A100 outperforms the RTX 3080 Ti by approximately 11%. Moreover, for the largest problem size evaluated, $N = 20\,000$, this advantage reaches 24.52%. Although the RTX 3080 Ti is also a powerful dGPU, it is designed primarily for desktop use, thus the superiority of the A100 is consistent with its HPC-oriented features. Following the only-dGPU executions are the two heterogeneous schemes that use them as accelerators, with the scheme involving the A100 performing the best, as expected.

The fact that, in the case of dGPUs, the accelerators deliver higher performance than the heterogeneous schemes is mainly due to the memory-bound nature of the problem and the characteristics of the devices, as discussed in detail in Sections 3.5.2 and 3.5.3. Next in the ranking are the various CPU models evaluated, with the Xeon Platinum 8352Y achieving the best performance, followed by the i9-12900K, and subsequently the Xeon Gold 6128 and the Xeon E-2176G. The Intel Xeon Platinum 8352Y features 32 physical cores and 64 threads, as well as 48 MB of Last Level Cache (LLC) cache. In this memory-bound scenario, the cache size plays a significant role in accelerating data access. In fact, its high computational capabilities and large LLC enable it to achieve competitive results in GFLOPS up to $N = 2048$, as shown in Figure 3.9. Following the Platinum, the Intel Core i9-12900K demonstrates strong performance, owing to its high base and boost clock frequencies, as well as support for up to 24 threads, despite having a smaller cache than the Platinum. The Xeon Gold 6128 comes next, offering 6 cores and 12 threads, similar to the Xeon E-2176G, but with a larger cache (19.25 MB vs. 12 MB L3), which provides a slight advantage in data reuse and latency reduction. Finally, the Xeon E-2176G, with its modest cache and memory bandwidth, is the slowest among the CPUs, delivering performance comparable to the CPU+iGPU and the iGPU executions. The performance of all the devices involved in the iGPU configuration is heavily affected by the shared memory bottleneck, as explained in Section 3.5.1. Meanwhile, the CPU+FPGA scheme consistently yields lower performance, mainly because it relies on a generic kernel that is not optimised for the FPGA's architecture. Despite outperforming the standalone FPGA across all sizes, the hybrid FPGA scheme is not better than the corresponding CPU-only execution for this problem.

Regarding the image denoising problem, the best execution times are achieved by the standalone NVIDIA A100, followed closely by the CPU+NVIDIA A100 configuration. These results reflect the compute-bound nature of the workload, which favours devices with high arithmetic throughput and extensive parallelism. The A100's architecture, optimised for high-performance workloads with massive floating-point capacity, proves to be particularly well-suited for this type of problem. Additionally, both the standalone A100 and its associated heterogeneous scheme achieve significantly higher GFLOPS compared to the other devices and configurations evaluated, as shown in Figure 3.10. The next best results are observed for the CPU+RTX 3080 Ti and the standalone RTX 3080 Ti, highlighting the strong computational capabilities of modern desktop-class GPUs. Subsequently, the CPUs Xeon Platinum and i9-12900K deliver good performance due to their previously mentioned high computational capabilities. Interestingly, these high-end CPUs outperform both the iGPU and the hybrid CPU+iGPU configurations. This fact can be mainly attributed to their superior scalar and SIMD throughput, large LLC that enhances data locality, low-latency memory hierarchies, and the absence of kernel launch overheads that typically affect GPU runtimes. Next come the hybrid CPU+iGPU approach and the standalone iGPU. As discussed in Section 3.5.1, the heterogeneous scheme achieves better performance by




(a) Heat diffusion.




(b) Image denoising.

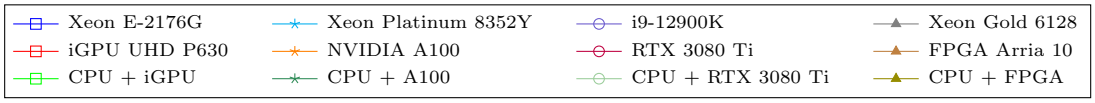



**Figure 3.8:** Execution times comparison for all the heterogeneous schemes and devices used in this study (lower is better). Note that there are no FPGA data beyond $N = 10\,000$, see explanation on Section 3.3.

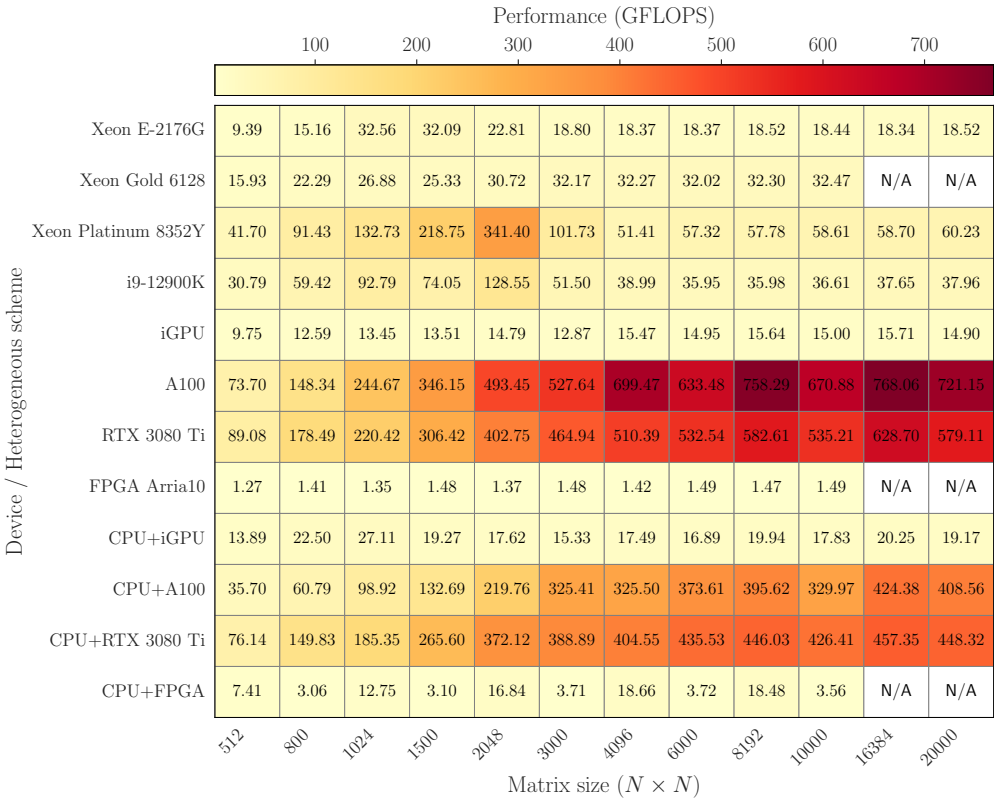



**Figure 3.9:** Real floating-point computational throughput in GFLOPS of the individual devices and the heterogeneous schemes evaluated for the heat diffusion problem across different matrix sizes.

offloading arithmetic-intensive operations to the iGPU; however, the shared memory bandwidth and the limited cache sizes of these systems constrain their efficiency. In this case, the Xeon E-2176G outperforms the Xeon Gold, likely due to its higher clock speed, which is advantageous for compute-dominated workloads. As with the heat diffusion problem, the FPGA configurations yield the worst performance. However, in this case, the CPU+FPGA scheme achieves better results than the CPU-only execution in some cases. Specifically, when the matrix dimensions are aligned with powers of two, as discussed in Section 3.5.4. This emphasises the advantage of workload sharing when host-device communication does not incur prohibitive overhead.

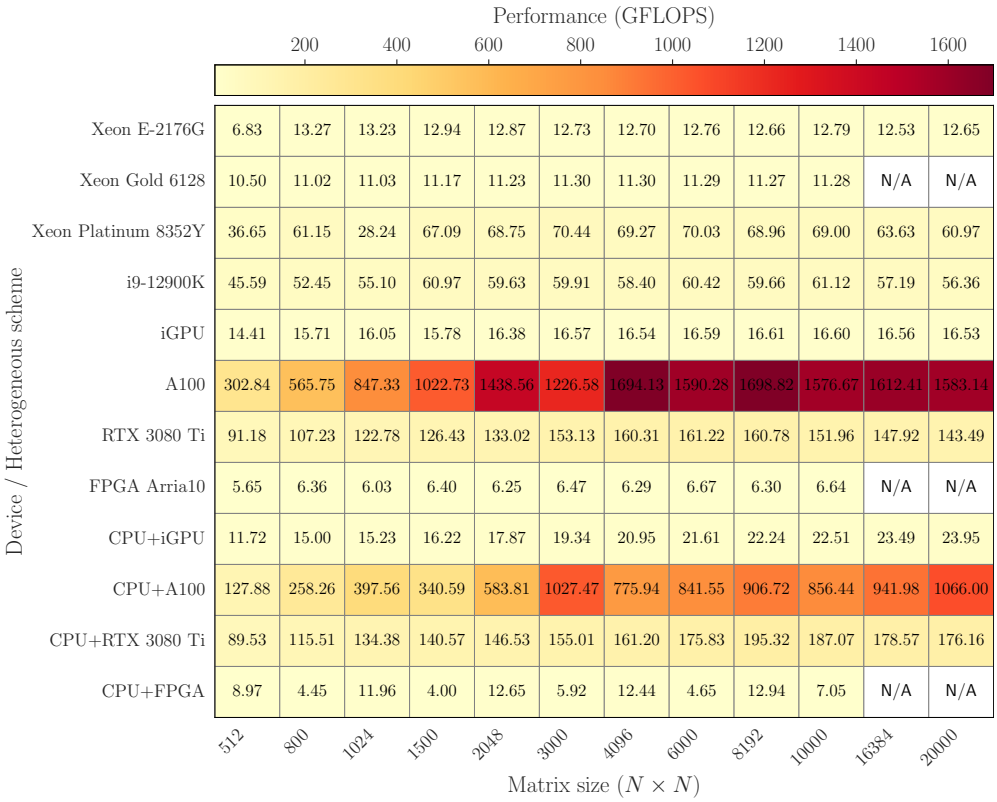



**Figure 3.10:** Real floating-point computational throughput in GFLOPS of the individual devices and the heterogeneous schemes evaluated for the image denoising problem across different matrix sizes.

## 3.6 Conclusions

This chapter assesses the feasibility of SYCL, specifically the Intel oneAPI implementation, for the development of portable heterogeneous computing solutions. To that end, a performance study was conducted on heterogeneous schemes for solving two representative iterative problems, heat diffusion and image denoising, using a dynamic workload balancing strategy at runtime. The analysis employed a range of devices, including Intel CPUs, GPUs with varying characteristics and vendors, and an Intel FPGA, as well as heterogeneous CPU+GPU and CPU+FPGA configurations. Furthermore, the experiments were carried out on two distributed computing systems, a cloud environment and a supercomputer, as well as a desktop machine.

For a memory-bound workload, represented by the heat diffusion problem, the

evaluated dGPUs offered the highest throughputs due to their superior memory bandwidth and parallel processing capabilities. Moreover, given their high efficiency, the time invested in rebalancing the load distribution and transferring data among devices was not offset by the use of a combined scheme. In contrast, the iGPU and its hybrid scheme provided limited benefits in this scenario, primarily due to shared memory constraints. Regarding the FPGA, as expected, the use of a suboptimal kernel for its architecture led to non-competitive results in terms of execution times. The CPU+FPGA scheme delivered modest gains over the individual device, with the best-case improvements relying heavily on efficient memory transfer patterns.

For a compute-bound workload, exemplified by the image denoising problem, the advantages of the heterogeneous configurations became more evident given the increased computational intensity. While iGPUs showed a better utilisation of its parallel architecture and consequently achieved improved results, the most significant performance gains were obtained with dGPUs. Particularly, the A100, which outperformed all the other devices by an order of magnitude. In this case, the CPU+dGPU scheme surpassed the GPU-only configuration when both devices contributed sufficiently to the overall computation, and their combination outweighed the overhead from communication and load rebalancing. Concerning the FPGA experiments, the results aligned with the previous case study. However, the heterogeneous configuration only exceeded the performance of the FPGA-only and CPU-only executions for matrix sizes that were powers of two, emphasising the necessity of efficient data transfers to process computationally intensive workloads.

Analysing the results, it can be concluded that the effectiveness of a heterogeneous scheme is closely linked to both the problem's computational characteristics and the underlying hardware architecture of the involved devices. In addition, dynamic load balancing proves essential to mitigate performance irregularities, particularly in schemes involving iGPUs and FPGAs.

Therefore, this study demonstrates the practicality of using SYCL, specifically Intel oneAPI's implementation, for unified heterogeneous computing development, facilitating rapid prototyping across diverse platforms. Nonetheless, it was observed that achieving optimal performance requires architecture-aware kernel design. This particularly affects FPGAs, precisely because their computational advantages stem from their reconfigurable architecture. To fully exploit this flexibility, it is necessary to use an ad-hoc implementation. Nevertheless, being able to integrate an FPGA into a heterogeneous system, even with a non-optimised kernel, significantly improves the programmability of the device when compared to the more traditional HDL-based approach discussed in Chapter 2. For CPUs and GPUs, kernel reuse is more feasible in embarrassingly parallel or SIMD-friendly workloads, such as the problems studied in this work. However, as expected, in contexts where both devices do not follow a similar strategy or cannot properly exploit their capacities (e. g., divergent control flow or irregular workloads), kernel reutilisation does not yield favourable results, as observed for the FPGA in this study. Another noteworthy observation is the varying

levels of integration and optimisation opportunities offered by high-level tools across the analysed devices. It is evident that CPUs and GPUs are at a more advanced stage of maturity compared to FPGAs, despite the rapid evolution of HLS tools in recent years. In any case, although the current state of HLS still leaves room for improvement, this high-level approach is highly promising for the democratisation of FPGAs usage as accelerators, facilitating efficient prototyping and enhanced integration.

Some of the most significant contributions of this chapter have been presented at the following conferences and published in the following journal:

- S. R. Alcaraz, R. Laso, O. G. Lorenzo, D. L. Vilariño, T. F. Pena, and F. F. Rivera, "Performance evaluation of heterogeneous CPU+iGPU and CPU+FPGA schemes using Intel OneAPI on the cloud", in *International Conference Computational and Mathematical Methods in Science and Engineering (CMMSE)*, Rota (Spain), 2023.
- S. R. Alcaraz, R. Laso, O. G. Lorenzo, D. L. Vilariño, T. F. Pena, and F. F. Rivera, "Assessing Intel OneAPI capabilities and cloud-performance for heterogeneous computing", *The Journal of Supercomputing*, vol. 80, pp. 13 295–13 316, 2024, ISSN: 1573-0484. DOI: 10.1007/s11227-024-05958-5. *'Reproduced with permission from Springer Nature'.*
- S. R. Alcaraz, R. Laso, D. L. Vilariño, and F. F. Rivera, "SYCL for CPU+GPU Heterogeneous Computing: A Study on Integrated and Discrete GPUs", in *2025 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*, 2025, pp. 1–2. DOI: 10.1109/CLUSTERWorkshops65972.2025.11164210. *'Reproduced with permission from © 2025 IEEE.'*

Chapter 4

# From CUDA to SYCL: Analysing automatic migration and performance portability on GPUs

*With a metric you can really go to town,*
*otherwise it is just abstract nonsense.*
— Jennifer T. Chayes, *MAA MathFest, 2007*

As introduced in previous chapters, the programmability and maintainability of heterogeneous applications can be cumbersome, since multiple devices often imply multiple code bases to develop and maintain. Multi-platform programming models, such as SYCL, have emerged as a solution to this issue by enabling portable implementations. Recently, migration tools have been developed to automatically translate code from widely used vendor-specific frameworks, such as CUDA, into these models. This chapter analyses the automatic migration capabilities of SYCLomatic by porting a well-known benchmark suite from CUDA to SYCL. Furthermore, the performance of the migrated SYCL and the original CUDA code are evaluated and compared, also including the calculation of the performance portability metrics introduced in Chapter 2.

## 4.1 A CUDA to SYCL case study

With the rise of heterogeneity in computing, the development of multi-platform programming tools has become especially important. The idea behind the automatic

migration tools, as those mentioned in Section 2.1.4, is to facilitate the transition from proprietary frameworks (e. g., CUDA) by automatically converting the code to open models (e. g., SYCL). Focusing on Intel's implementation of SYCL, Intel oneAPI, the book on DPC++ dedicates a whole chapter to outlining the key differences between CUDA and SYCL, highlighting important considerations when migrating code, including when relying on automated solutions [231].

The tool used in this work to migrate CUDA code to SYCL is SYCLomatic, which is part of the Intel oneAPI Base Toolkit under its commercial name DPCT. Specifically, the 2025.1.0 release of the toolkit was employed. To enable execution of the migrated SYCL code on NVIDIA GPUs, the Codeplay plugin for NVIDIA devices was used. This plugin follows the same versioning scheme as the Intel oneAPI toolkit, and the 2025.1.0 release was used for both to ensure compatibility. The base CUDA code corresponds to the Rodinia benchmark suite [232], which serves as a suitable example due to its recognition and inclusion of widely used algorithms. This makes it appropriate for comparing the performance and portability of CUDA and the automatically migrated SYCL.

### 4.1.1 SYCLomatic

SYCLomatic is an open-source migration tool hosted on GitHub [20] and developed by Intel to facilitate the conversion of CUDA-based C++ code to DPC++. It is released under the Apache 2.0 licence with the Low Level VM (LLVM) exception, which permits redistribution of compiler-generated codes without additional attribution and ensures compatibility with GPLv2. According to Intel, the tool is designed to automate approximately 90 to 95% of the translation process from CUDA kernels and API calls to SYCL [233]. The output is human-readable DPC++ code annotated with diagnostic messages and warning codes [234] to help developers understand the translation and identify segments requiring manual intervention. The overall migration workflow is illustrated in Figure 4.1. Starting from CUDA source code, SYCLomatic automatically transforms it into SYCL-compliant syntax, embedding inline annotations to flag constructs that may require manual review. The resulting code is then compiled, optimised for the target architecture, and deployed on the selected accelerator.

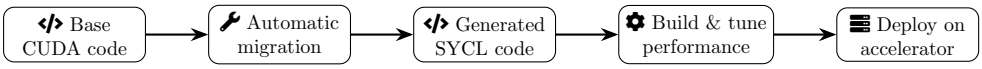



**Figure 4.1:** Workflow employed by SYCLomatic to translate CUDA source code into SYCL-based C++ and deploy it on heterogeneous devices.

Concerning the migration process, it relies primarily on a Clang-based front end, a set of migration rules, the DPCT compatibility library with helper APIs, diagnostics

and comments, and a collection of build and migration support scripts. SYCLomatic uses the Clang compiler to analyse CUDA source code and produce an internal representation of its structure and semantics. Although the official documentation does not explicitly confirm it, most source-to-source translation tools rely on Abstract Syntax Trees (ASTs) to enable context-aware transformations, which are more accurate than simple syntactic substitutions between CUDA and SYCL. At the core of SYCLomatic is its set of migration rules, which define how CUDA constructs are translated into their SYCL counterparts. This study employs the default rule database provided by the tool, though custom rule sets can also be defined. The header-only DPCT compatibility library offers replacements for CUDA features that have no direct equivalents in SYCL. Consequently, the migrated code often includes references to the `dpct::` namespace for device selection, queue management and other utilities. The diagnostic system adds in-line comments to the migrated code, including warnings that help developers understand the translation and identify areas that may require manual adjustments. SYCLomatic also provides scripts to assist with the migration of complete projects, including those using build systems such as CMake or Makefiles.

Recapping, the automatic steps performed by the tool include analysing the base CUDA code, identifying known patterns, and applying the relevant migration rules. After this step, the tool generates the migrated SYCL code along with diagnostic comments that help interpret the required manual adjustments as well as the applied transformations.

### 4.1.2 Rodinia benchmark suite

The benchmark suite used in this work to perform the CUDA to SYCL migration is Rodinia [232], a widely recognised suite designed specifically for heterogeneous computing, and publicly available on Github [235]. It provides a diverse set of compute-intensive applications targeting multi-core CPUs and GPUs with implementations in OpenMP, OpenCL, and CUDA. Table 4.1 summarises the set of applications included in the benchmark suite, along with a description of each program's domain and the available implementations. The selection of applications is inspired by Berkeley's dwarf taxonomy [236], which identifies a set of fundamental computational patterns common across a broad spectrum of applications. This ensures that Rodinia covers a wide range of parallel communication patterns, synchronisation methods, and computational behaviours. The latest official release of Rodinia is version 3.1, dating back to 2014, and it no longer receives official support. Nevertheless, it remains a widely used benchmark within the community due to its broad selection of algorithms and long-standing role in heterogeneous computing performance evaluation. Given its development period, the implementations include CUDA versions ranging from 4.0 to 8.0. Consequently, some of the code had to be adapted to be compatible with the CUDA version used in this work, which was 12.2. Moreover, it was necessary to in-

clude time measurements in some benchmarks and adapt others to ensure a common format, thereby facilitating the analysis and comparison of results.

To improve the reproducibility of this study, the code used to conduct the experiments is publicly accessible on GitHub[1]. The input data and the parameters used for each application can also be found in the corresponding *run* files.

**Table 4.1:** Applications included in the Rodinia benchmark suite.

| Application | Workload | Domain | Implementations |
|---|---|---|---|
| b+tree | Graph Traversal | Search | CUDA, OMP, OCL |
| backprop | Unstructured Grid | Pattern Recognition | CUDA, OMP, OCL |
| bfs | Graph Traversal | Graph Algorithms | CUDA, OMP, OCL |
| cfd | Unstructured Grid | Fluid Dynamics | CUDA, OMP, OCL |
| dwt2d | Spectral Method | Image/Video Compression | CUDA, OCL |
| gaussian | Dense Linear Algebra | Linear Algebra | CUDA, OCL |
| heatwall | Structured Grid | Medical Imaging | CUDA, OMP, OCL |
| hotspot | Structured Grid | Physics Simulation | CUDA, OMP, OCL |
| hotspot3D | Structured Grid | Physics Simulation | CUDA, OMP, OCL |
| huffman | Finite State Machine | Lossless data compression | CUDA, OCL |
| hybridsort | Sorting | Sorting Algorithms | CUDA, OCL |
| kmeans | Dense Linear Algebra | Data Mining | CUDA, OMP, OCL |
| lavaMD | N-Body | Molecular Dynamics | CUDA, OMP, OCL |
| leukocyte | Structured Grid | Medical Imaging | CUDA, OMP, OCL |
| lud | Dense Linear Algebra | Linear Algebra | CUDA, OMP, OCL |
| mummergpu | Graph Traversal | Bioinformatics | CUDA, OMP |
| myocyte | Structured Grid | Biological Simulation | CUDA, OMP, OCL |
| nn | Dense Linear Algebra | Data Mining | CUDA, OMP, OCL |
| nw | Dynamic Programming | Bioinformatics | CUDA, OMP, OCL |
| particlefilter | Structured Grid | Medical Imaging | CUDA, OMP, OCL |
| pathfinder | Dynamic Programming | Grid Traversal | CUDA, OMP, OCL |
| srad | Structured Grid | Image Processing | CUDA, OMP, OCL |
| streamcluster | Dense Linear Algebra | Data Mining | CUDA, OMP, OCL |

### 4.1.3 Background on CUDA to SYCL migration

The practical viability of SYCLomatic for migrating CUDA code to SYCL has been explored across various domains. In the field of biotechnology, Costanzo et al. [237–239] effectively demonstrated the ease of porting specific CUDA-based applications to SYCL using Intel oneAPI. Jin's study [240] identified challenges in the migration

[1] https://github.com/silviaralcaraz/syclomatic-eval

process arising from differences in feature support between CUDA and SYCL, which may hinder a smooth transition, especially for complex applications such as parallel graph colouring. Faqir-Rhazoui et García [241] reported comparable performance between migrated SYCL and CUDA on NVIDIA GPUs, although the SYCL code was limited in memory optimisations due to compiler restrictions. Following a similar direction, Weckert et al. [242] presented Altis-SYCL, a benchmark suite migrated from CUDA using the DPCT tool, demonstrating competitive performance across both GPUs and FPGAs. Similarly, Jadhav et al. [243] migrated a CUDA-based seismic simulation code to SYCL, achieving performance comparable to CUDA on NVIDIA GPUs and a notable speedup on Intel GPUs. In the domain of astrophysics, Liang et al. [218] described the migration of the GPU-accelerated component of the ARGOT simulation code to Intel oneAPI using SYCLomatic.

Previous work on migrating the Rodinia benchmark from CUDA to SYCL using DPCT was presented by Castaño et al. [244], who reported around 87% of successfully migrated code. They also evaluated SYCL against CUDA on NVIDIA GPUs, executed SYCL on Intel GPUs for comparison, and examined SYCL alongside OpenMP on Intel CPUs. This work can be considered as an update of the study by Castaño et al., focusing solely on GPUs and enhancing their analysis with the calculation of performance portability metrics to provide a more accurate assessment. Moreover, it enables a comparison of the evolution of the tools used, as Castaño et al. conducted their research with oneAPI v2021.4 and CUDA 11.4, relying on experimental support for NVIDIA GPUs. Finally, the decision to focus exclusively on GPUs is motivated by the fact that, using CUDA code as a basis, the resulting SYCL implementation is inherently GPU-oriented, making performance and portability evaluations on these devices more consistent.

Together, the aforementioned works illustrate the growing maturity and potential of automatic migration tools for heterogeneous computing. However, the collective evidence suggests that while significant progress has been made, programmer intervention is often required to deal with inefficiencies and runtime errors arising from inherent differences in execution models.

## 4.2 Results and discussion

This section presents the experimental results along with their corresponding analysis. First, the base code was preliminarily examined using SYCLomatic's diagnostic tools. Second, the degree of automation was assessed based on the success of the migration and the warnings generated during the process. Third, the performance of both the original CUDA code and the migrated SYCL implementation was evaluated, including comparisons of the execution times and the computation of the performance portability metrics.

### 4.2.1 Preliminary analysis

A preliminary analysis was conducted using the `--analysis-mode` option of the SYCLomatic tool prior to the migration process. This step aimed to evaluate the feasibility of automatic migration by identifying CUDA constructs that may require manual intervention. Specifically, SYCLomatic classifies code constructs at the line level based on their expected migration complexity into Migration Effort (ME) categories: "No-ME", for constructs fully supported by automatic translation; "Low-ME", for those requiring minor manual changes; "Medium-ME", for those requiring moderate manual changes; and "High-ME", for cases where extensive manual intervention is anticipated.

To systematically apply this analysis across the Rodinia benchmark suite, a custom Bash script was developed to traverse each application of the CUDA implementation, regenerate the build databases using the `intercept-build` command, and invoke `dpct --analysis-mode` on all relevant source files. The resulting reports were consolidated into a single output for inspection, facilitating the generation of the Table 4.2, which summarises the analysis results across the 24 benchmarks of the Rodinia suite.

**Table 4.2:** Preliminary study using SYCLomatic analysis mode.

| Algorithm | Total CUDA lines | No-ME (APIs/types) | Low-ME (APIs/types) | Medium-ME (APIs/types) | High-ME (APIs/types) | Auto-Migratable (%) |
|---|---|---|---|---|---|---|
| b+tree | 59 | 52 | 3 | 4 | 0 | 100 |
| backprop | 35 | 23 | 11 | 1 | 0 | 100 |
| bfs | 27 | 23 | 4 | 0 | 0 | 100 |
| cfd | 153 | 86 | 67 | 0 | 0 | 100 |
| dwt2d | 100 | 95 | 4 | 1 | 0 | 100 |
| gaussian | 25 | 16 | 8 | 1 | 5 | 83 |
| heatwall | 101 | 64 | 3 | 34 | 0 | 100 |
| hotspot | 14 | 9 | 3 | 2 | 0 | 100 |
| hotspot3D | 11 | 10 | 1 | 0 | 0 | 100 |
| huffman | 117 | 86 | 20 | 11 | 1 | 99 |
| hybridsort | 84 | 46 | 17 | 21 | 0 | 100 |
| kmeans | 19 | 14 | 4 | 1 | 0 | 100 |
| lavaMD | 21 | 16 | 2 | 3 | 0 | 100 |
| leukocyte | 67 | 45 | 14 | 8 | 0 | 100 |
| lud | 16 | 4 | 10 | 2 | 0 | 100 |
| mummergpu | 84 | 70 | 9 | 5 | 0 | 100 |
| myocyte | 315 | 308 | 7 | 0 | 0 | 100 |
| nn | 14 | 11 | 2 | 1 | 0 | 100 |
| nw | 23 | 7 | 12 | 4 | 0 | 100 |
| particlefilter | 85 | 69 | 14 | 2 | 0 | 100 |
| pathfinder | 16 | 11 | 3 | 2 | 0 | 100 |
| srad v1 | 53 | 36 | 14 | 3 | 0 | 100 |
| srad v2 | 33 | 15 | 18 | 0 | 0 | 100 |
| streamcluster | 69 | 65 | 4 | 0 | 0 | 100 |

Notably, 22 benchmarks were identified as 100% auto-migratable, containing only No-ME, Low-ME, or Medium-ME constructs. This suggests that SYCLomatic should

be capable of handling most of the translation with minimal manual adjustment. The estimated percentage of code that is automatically migratable is also reported by the tool and, according to the results, appears to depend primarily on the presence or absence of "High-ME" lines. The only two benchmarks flagged with higher migration effort were *gaussian* and *huffman*. *Gaussian*, in particular, includes five "High-ME" patterns, while *huffman* contains one. These findings suggest that while most benchmarks are suitable candidates for automated migration, these two are likely to present challenges that require substantial manual refactoring. In any case, although *mummergpu* did not exhibit notable issues during the initial analysis, it proved to be strongly incompatible with the modern CUDA version used in this study. As previously noted in Section 4.1.2, certain benchmarks required modifications to ensure compatibility. These included, among others, the replacement of constructs no longer supported by modern C standards and the adaptation of texture usage to the current CUDA API. In the case of *mummergpu*, however, the required changes were not straightforward and demanded a more extensive reimplementation, which lay outside the scope of this work. Consequently, it was excluded from the evaluation.

### 4.2.2 Level of automation

To assess the level of automation achieved by the tool, the success of the migration was analysed across all benchmarks (23 applications, excluding *mummergpu*). Only *hybridsort* and *kmeans* could not be directly migrated and required manual intervention. In this context, direct migration refers to the ability to execute the generated code without modifying the source, requiring only changes to the compilation options to target DPC++ with a CUDA back-end instead of NVCC. It is worth noting that the two non-migrated programs do not correspond to those identified as the most problematic in the initial analysis, as will be discussed in the subsequent section. In total, 1421 CUDA lines were migrated, of which 103 correspond to the two applications that could not be directly migrated, resulting in an overall migration success rate of 93.24%. If *mummergpu* had been taken into account, the total number of lines would be 1524, with 187 non-migrated CUDA lines, leading to a success rate of 88.38%. Additionally, although *leukocyte* was successfully migrated and compiled without errors, its execution failed. This failure was due to the lack of full support for SYCL 1.2.1 image objects on the evaluated devices when using the CUDA back-end, even with the `SYCL_PI_CUDA_ENABLE_IMAGE_SUPPORT` variable enabled, which only provides partial functionality. As this issue arises from a feature not supported by the CUDA back-end rather than from a problem in the migration process, the case was still classified as a successful migration. However, for this reason, *leukocyte* had to be excluded from the performance evaluation. The level of automation achieved is consistent with the rate reported by Intel and similar to that achieved by Castaño et al. [244]. These results suggest that the improvements introduced in the tool over

recent years have had a limited impact on the software under analysis, which is expected given its release date and the absence of modern features. Nevertheless, the overall degree of automatic migration remains high and approaches a fully automated process.

#### 4.2.2.1 Analysis of the migration warnings

In order to facilitate the analysis of the warnings generated during the automatic migration, all the predefined warnings detected by the tool were grouped into semantically coherent categories, each corresponding to a specific type of migration challenge. While some warnings are purely informative, others indicate potential issues such as semantic mismatches, incomplete translations, or areas requiring manual intervention. Therefore, this process involved analysing the description of each warning, focusing on recurring patterns such as references to unsupported APIs, memory model changes, performance implications, or explicit mentions of manual rewrites.

The classification of the set of SYCLomatic warnings resulted in the following ten groups, with the corresponding percentage of the total attributed to each category: *manual_fixes*, which refers to situations where the tool provides incomplete migration and requires the developer to review and complete the translation manually (20.61%); *unsupported*, which includes features or constructs that have no SYCL equivalent and therefore cannot be migrated without redesign or removal (19.85%); *correctness*, which indicates potential deviations in numerical accuracy or behaviour between the original and the migrated code, requiring thorough validation (16.79%); *API_differences*, related to mismatches in API semantics or availability of SYCL compared to CUDA (13.74%); *unresolved*, where the tool is unable to translate code segments and leaves it untranslated (11.45%); *error_handling*, involving changes in exception or error management (6.11%); *memory*, which concerns allocation and pointer usage differences (5.34%); *performance*, highlighting possible slowdowns due to altered runtime behaviour or algorithmic structure (3.82%); *language_limitation*, which stems from SYCL or C++ restrictions that prevent the direct translation of certain constructs, often requiring refactoring (1.53%); and *time_measurement*, which reflects the lack of support for CUDA-style timing APIs in SYCL and thus demands alternative strategies (0.76%).

Table 4.3 summarises the resulting classification, listing each group alongside the corresponding warning identifiers and the percentage of total warnings attributed to that category. These percentages reflect the relative prevalence and, by extension, the potential practical significance of each class of migration issue. This information facilitates the interpretation of the results, since a higher proportion of warnings in a given category does not necessarily imply greater migration complexity. In some cases, it may simply indicate that the tool is more likely to detect and report certain types of patterns, depending on the predefined set of warnings it is designed to recognise.

Finally, note that a complete description of each warning is available in Intel's official guide [234].

**Table 4.3:** Classification of SYCLomatic warnings into groups, including the percentage of total warnings attributed to each category and the corresponding warning identifiers.

| Group | Percentage | Warning IDs |
|---|---|---|
| manual_fixes | 20.61% | DPCT1016, DPCT1033, DPCT1037, DPCT1050, DPCT1054, DPCT1055, DPCT1056, DPCT1057, DPCT1058, DPCT1059, DPCT1060, DPCT1068, DPCT1070, DPCT1073, DPCT1079, DPCT1089, DPCT1090, DPCT1091, DPCT1094, DPCT1095, DPCT1103, DPCT1104, DPCT1122, DPCT1123, DPCT1128, DPCT1130, DPCT1131 |
| unsupported | 19.85% | DPCT1007, DPCT1008, DPCT1014, DPCT1020, DPCT1021, DPCT1023, DPCT1029, DPCT1030, DPCT1031, DPCT1046, DPCT1051, DPCT1052, DPCT1053, DPCT1062, DPCT1072, DPCT1074, DPCT1082, DPCT1087, DPCT1100, DPCT1102, DPCT1106, DPCT1107, DPCT1108, DPCT1119, DPCT1127, DPCT1132 |
| correctness | 16.79% | DPCT1013, DPCT1015, DPCT1022, DPCT1032, DPCT1038, DPCT1047, DPCT1048, DPCT1064, DPCT1075, DPCT1077, DPCT1083, DPCT1096, DPCT1097, DPCT1098, DPCT1099, DPCT1101, DPCT1105, DPCT1111, DPCT1112, DPCT1120, DPCT1121, DPCT1129 |
| API_differences | 13.74% | DPCT1005, DPCT1006, DPCT1011, DPCT1025, DPCT1035, DPCT1040, DPCT1042, DPCT1043, DPCT1045, DPCT1049, DPCT1061, DPCT1063, DPCT1066, DPCT1085, DPCT1086, DPCT1093, DPCT1113, DPCT1118 |
| unresolved | 11.45% | DPCT1001, DPCT1004, DPCT1026, DPCT1027, DPCT1028, DPCT1036, DPCT1044, DPCT1067, DPCT1069, DPCT1071, DPCT1076, DPCT1084, DPCT1088, DPCT1125, DPCT1126 |
| error_handling | 6.11% | DPCT1000, DPCT1002, DPCT1003, DPCT1009, DPCT1010, DPCT1024, DPCT1034, DPCT1041 |
| memory | 5.34% | DPCT1019, DPCT1039, DPCT1078, DPCT1081, DPCT1114, DPCT1115, DPCT1124 |
| performance | 3.82% | DPCT1017, DPCT1018, DPCT1065, DPCT1092, DPCT1110 |
| language_limitation | 1.53% | DPCT1080, DPCT1109 |
| time_measurement | 0.76% | DPCT1012 |

After the migration of the full Rodinia benchmark suite, a total of 471 warnings were generated. Categories such as *API_differences* (35.24%), *performance* (27.81%), and *error_handling* (17.41%) were the most frequently reported across the benchmarks, as depicted in Figure 4.2. Thus, in this migration, the most prevalent groups did not correspond to those with the highest number of cases or warnings listed in Table 4.3.

In addition, Figure 4.3 presents the distribution of warnings per benchmark, grouped by category. The applications with the highest number of warnings are *heartwall* and *cfd*, indicating that their migration involves a wider variety of constructs that trigger diagnostic messages. As previously mentioned, in this case study, when considering

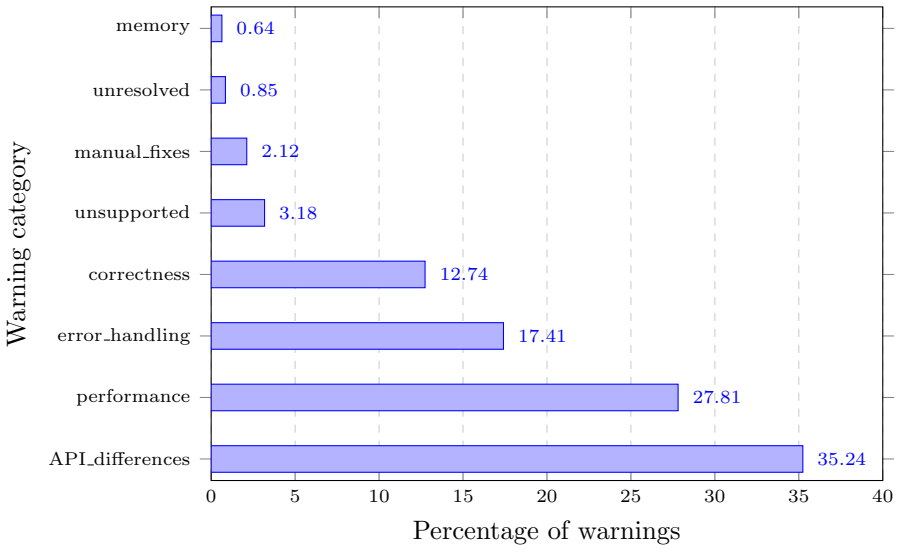



**Figure 4.2:** Distribution of SYCLomatic warning categories across the complete Rodinia benchmark suite, ordered by decreasing frequency.

both the overall success of the migration and the warnings generated, the results diverge from the initial analysis, which had identified *gaussian* and *huffman* as the most challenging applications. Given that the migration of the entire benchmark suite required only about one minute on the machine described in Appendix A.5, the findings suggest that, although preliminary analyses may assist in estimating code complexity, it remains worthwhile to perform a complete migration. Furthermore, the structural complexity of the original code does not appear to be a decisive factor in migration success, as certain complex constructs may have direct equivalents in SYCL.

To investigate further the relation between warning incidence and the applications under study, the association between the number of CUDA lines in each programme and the warnings generated were examined, resulting in the scatter plot depicted in Figure 4.4. The Pearson coefficient ($r = 0.33$) indicates a weak-to-moderate positive correlation. In general, the applications comprising more CUDA code tend to generate more warnings during the migration, although the trend is weak and several observations deviate from the regression line. Moreover, the p-value ($p = 0.13$) exceeds the conventional 0.05 threshold, demonstrating a statistically non-significant relationship. Consequently, only about 11% of the variance in migration warning counts ($r^2 \approx 0.11$) can be attributed to the code size, while the fan-shaped residual pattern and a single high-leverage outlier further violate linearity assumptions. These findings demonstrate that code volume alone is an unreliable predictor of warning generation in CUDA to SYCL migration. Therefore, the size of the code does not appear to be a critical factor

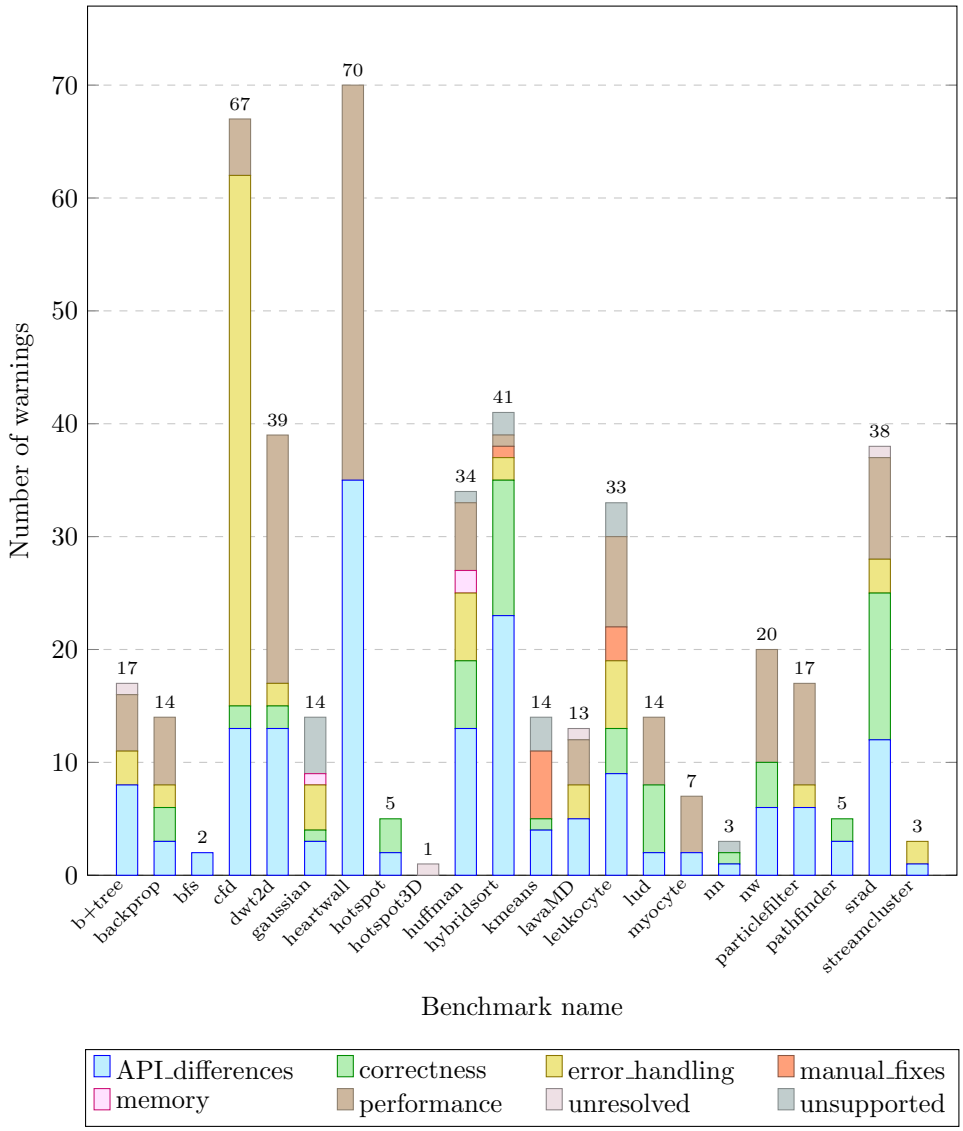



**Figure 4.3:** Total number of SYCLomatic warnings per benchmark, grouped by category. Labels above each bar indicate the overall number of warnings generated during the migration.

when performing the migration, nor does it necessarily affect its success.

Based on the statistical analysis conducted, the success of the migration and the generation of warnings by SYCLomatic do not appear to depend directly on either the dimensions of the code or the complexity of its structures. Rather, these outcomes seem to be more closely related to factors such as the availability of suitable translations in SYCL, the use of supported features, specific code patterns, and other similar aspects.

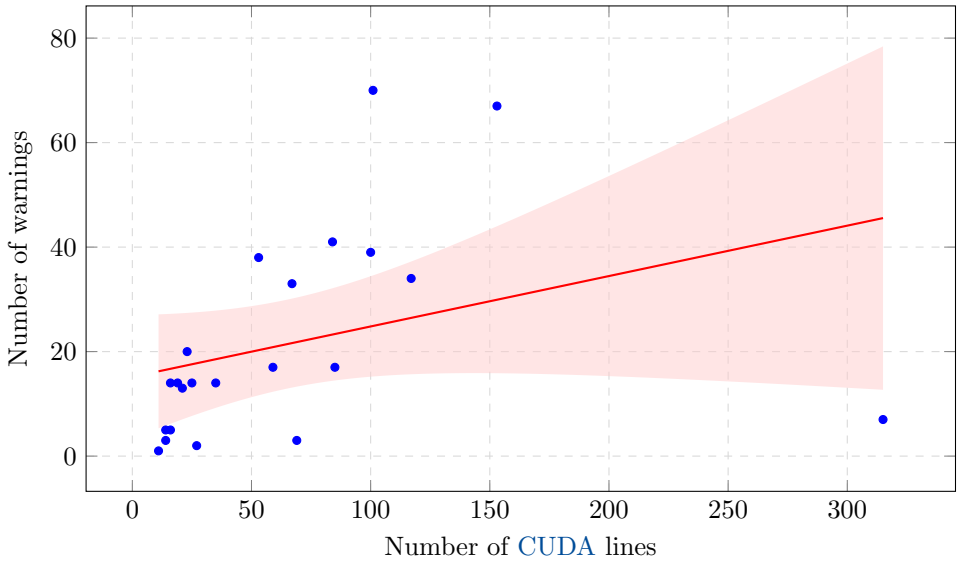



**Figure 4.4:** Scatter plot of the number of CUDA source-lines versus the number of migration warnings for each benchmark. The red line represents the linear-regression fit, with its 95% confidence interval shaded. Pearson's correlation coefficient is $r = 0.33$ (two-tailed $p = 0.13$).

### 4.2.3 Performance evaluation on HPC GPUs

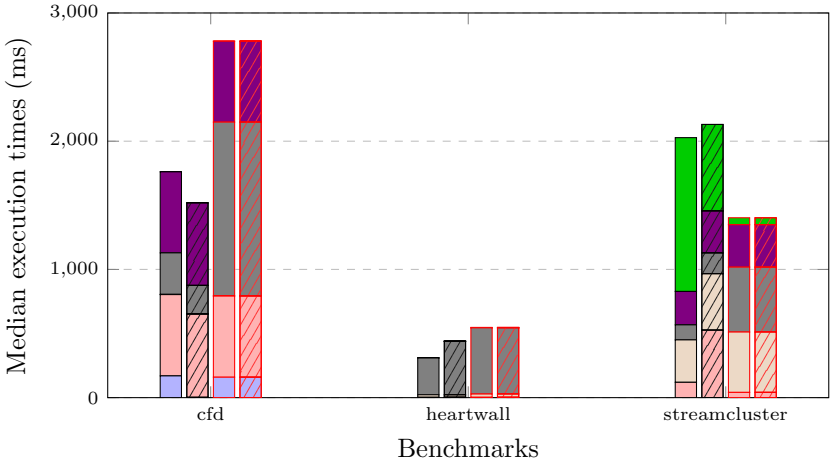
This section presents an analysis of the execution times of the two implementations under study, namely CUDA and the automatically migrated SYCL, along with an assessment of their performance portability in this case study through the calculation of the $\Phi$ and $\overline{\Phi}$ metrics. Specifically, the execution times correspond to the median of ten measurements, preceded by five warm-up iterations that were discarded to eliminate potential initialisation overheads.

#### 4.2.3.1 Comparison of the execution times

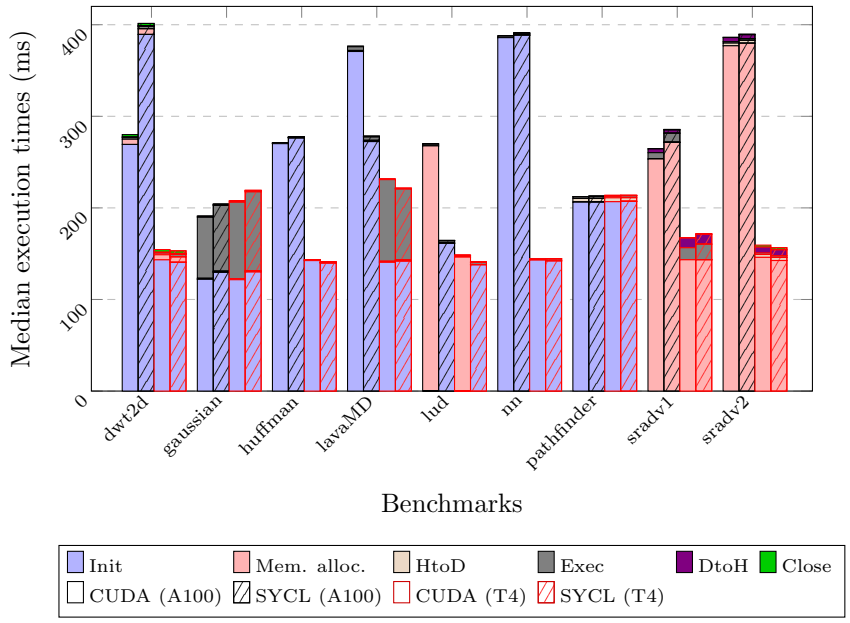
The experiments compare the execution times obtained with CUDA and SYCL for the set of Rodinia applications, excluding those discarded or not automatically migrated (*hybridsort*, *kmeans*, *leukocyte*, and *mummergpu*). The evaluation was conducted on two NVIDIA GPU nodes of the FT3 supercomputer, equipped with an A100 and a Tesla T4, respectively. Node characteristics are detailed in Appendices A.3 and A.4.

To improve the visualisation of the results, the benchmarks were divided into four groups based on their execution times measured in milliseconds. This grouping allows meaningful comparisons without distortion from differences in scale, concretely: "Group 1", Figure 4.5(a), includes applications with execution times $\geq 500$ ms; "Group 2", Figure 4.5(b), between 150 ms and 500 ms; "Group 3", Figure 4.6(a), between 10 ms and 150 ms; and "Group 4", Figure 4.6(b), $\leq 10$ ms. The median execution times presented in the figures consider only the portions of the code under study to enhance comparability, including: initialisation routines (*init*), memory allocations (*Mem. alloc.*), host-to-device communications (*HtoD*), kernel execution (*Exec*), device-to-host communications (*DtoH*), and memory deallocations (*close*).

Most of the benchmarks provide similar execution times when comparing the migrated SYCL and the native CUDA. It should be noted that the SYCL execution of *backprop* failed on the Tesla T4 due to a runtime CUDA error not experienced on the A100. The applications that exhibit the highest differences are *dwt2d*, *lavaMD*, and *lud* on the A100, and *hotspot3D* and *particlefilter-float* on the T4. In some of these cases, for applications running on the A100, it appears that the CUDA code itself suffers from long initialisation or memory allocation times. Therefore, the SYCL implementation possibly amplifies these problems with extra calls to the CUDA API and other additional mechanisms. In the remaining two cases, *hotspot3D* and *particlefilter-float*, longer kernel execution times are observed for CUDA on both the A100 and the T4, with the effect more pronounced on the latter. These types of differences indicate the need for in-depth profiling, as the consistency of final performance when migrating from one application to another can fluctuate with both the hardware and the specific problem. To further examine performance across hardware, the results for each GPU were analysed individually. Some applications (*pathfinder*, *hotspot*, *nw*, *myocyte*, *particlefilter-naive*) exhibited very similar performance on both platforms. Among the remaining applications, half achieved better performance on the A100, while the other half performed better on the T4. For *cfd*, *heartwall*, *gaussian*, and *hotspot3D*, kernel execution times were longer on the T4, likely reflecting the greater computational capabilities of the A100. In other cases, including *dwt*, *huffman*, *nn*, and *srad*, the T4 performed better due to longer initialisation and memory allocation times on the A100. Although the A100 is a more powerful GPU in terms of theoretical peak performance and memory bandwidth, the Rodinia benchmarks with the selected parameters and inputs are relatively lightweight and do not fully utilise its resources, which may explain the observed results.

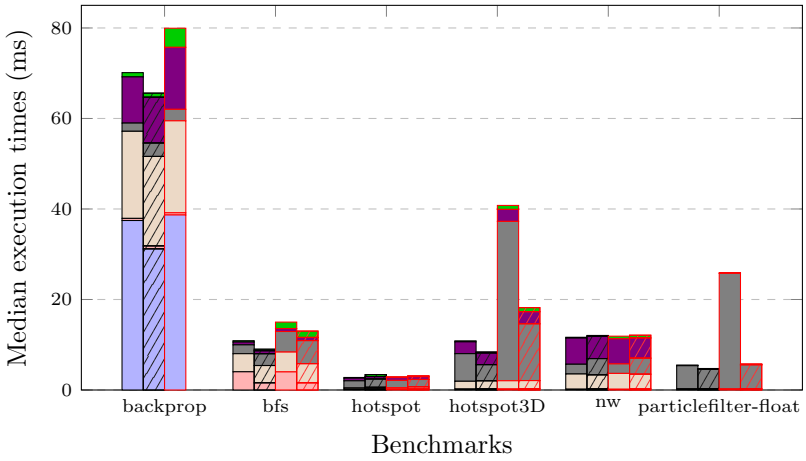



(a) Group 1: benchmarks with median execution times $\geq$ 500 ms.

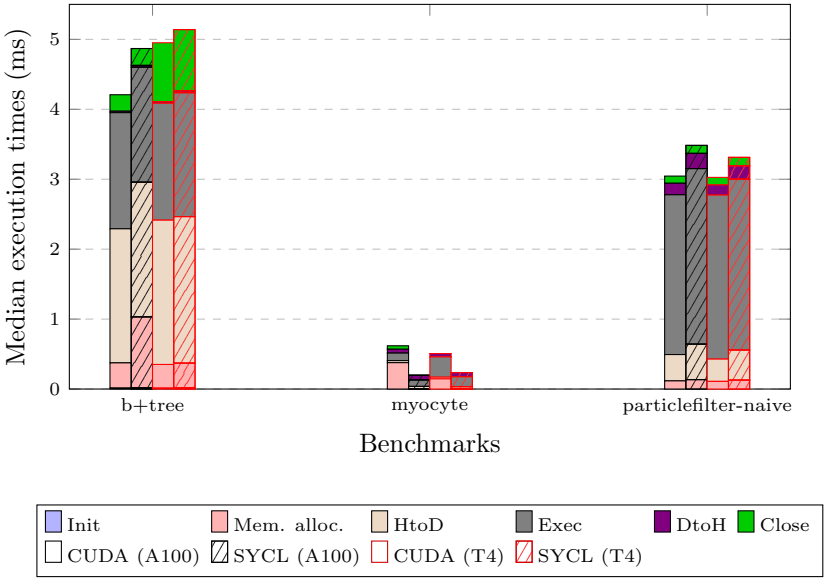



(b) Group 2: benchmarks with median execution times between 150 ms and 500 ms.

**Figure 4.5:** Median execution times for benchmark groups 1 and 2, comparing CUDA and SYCL implementations on the NVIDIA A100 and Tesla T4 GPUs.




(a) Group 3: benchmarks with median execution times between 10 ms and 150 ms.




(b) Group 4: benchmarks with median execution times $\leq$ 10 ms.

**Figure 4.6:** Median execution times for benchmark groups 3 and 4, comparing CUDA and SYCL implementations on the NVIDIA A100 and Tesla T4 GPUs.

#### 4.2.3.2 Performance portability

To evaluate performance portability, the $\Phi$ and $\overline{\Phi}$ metrics introduced in Chapter 2 have been calculated. The first, as it remains the standard and most widely used metric in this type of study, the second, due to its recent adoption, mainly for evaluating application efficiency, as is the case.

Formalising the problem, the study employs the platform set $\mathcal{H} = \{\text{A100}, \text{Tesla T4}\}$, the applications CUDA and SYCL, and the problems corresponding to the benchmarks of the Rodinia suite. Table 4.4 presents the computed values of both metrics for CUDA and SYCL across each problem. In addition, Appendix B.2 provides all the data required to reproduce the results.

**Table 4.4:** Performance portability scores for the metrics $\Phi$ and $\overline{\Phi}$ with the Rodinia benchmarks for the evaluated frameworks, CUDA and SYCL.

| | CUDA | | SYCL | |
|---|---|---|---|---|
| **Benchmark** | $\Phi$ | $\overline{\Phi}$ | $\Phi$ | $\overline{\Phi}$ |
| b+tree | 0.9535 | 0.9555 | 0.9232 | 0.9286 |
| backprop | 0.6066 | 0.7177 | 0.0000 | 1.0000 |
| bfs | 0.9989 | 0.9989 | 0.9979 | 0.9979 |
| cfd | 0.8048 | 0.8367 | 0.7887 | 0.8256 |
| dwt2d | 0.7301 | 0.7875 | 0.5720 | 0.7003 |
| gaussian | 0.9814 | 0.9818 | 0.9112 | 0.9185 |
| heartwall | 0.8936 | 0.9038 | 0.9487 | 0.9512 |
| hotspot | 0.8939 | 0.9041 | 0.7005 | 0.7695 |
| hotspot3D | 0.9911 | 0.9912 | 0.9890 | 0.9891 |
| huffman | 0.6564 | 0.7443 | 0.6908 | 0.7638 |
| hybridsort | 0.7953 | 0.8301 | 0.0000 | 0.0000 |
| kmeans | 0.9974 | 0.9974 | 0.0000 | 0.0000 |
| lavaMD | 0.7620 | 0.8077 | 0.8868 | 0.8983 |
| leukocyte | 0.7062 | 0.7729 | 0.0000 | 0.0000 |
| lud | 0.7095 | 0.7749 | 0.9231 | 0.9286 |
| myocyte | 0.9420 | 0.9452 | 0.9114 | 0.9186 |
| nn | 0.5664 | 0.6975 | 0.5587 | 0.6938 |
| nw | 0.6342 | 0.7322 | 0.9968 | 0.9968 |
| particlefilter_float | 0.7722 | 0.8145 | 0.7213 | 0.7821 |
| particlefilter_naive | 0.7085 | 0.7743 | 0.7121 | 0.7765 |
| pathfinder | 0.9458 | 0.9486 | 0.9496 | 0.9520 |
| srad_v1 | 0.7857 | 0.8235 | 0.6333 | 0.7317 |
| srad_v2 | 0.7110 | 0.7758 | 0.6811 | 0.7582 |
| streamcluster | 0.9212 | 0.9270 | 0.8648 | 0.8809 |

Regarding the interpretation of the results, the individual values obtained for each benchmark were grouped by programming model, and the geometric mean was then computed for both CUDA and SYCL across the two metrics. The results reveal that SYCL achieves slightly higher values than CUDA in both scenarios. Considering all the benchmarks, SYCL attains $\Phi = 0.8048$ and $\overline{\Phi} = 0.8586$, compared with $\Phi = 0.8000$ and $\overline{\Phi} = 0.8465$ for CUDA. When excluding the cases of *hybridsort*, *kmeans*, and *leukocyte*, the geometric mean remains the same for SYCL, while CUDA reaches $\Phi = 0.7966$ and $\overline{\Phi} = 0.8443$. Therefore, SYCL offers a marginal yet consistent advantage in performance portability across the Rodinia suite.

Concerning the differences between the metrics, $\Phi$ applies a stronger penalty to imbalances across platforms. This effect can be observed in applications such as *dwt2d*, *huffman*, and *nn*, where more pronounced differences between devices result in a noticeable reduction in the metric score. Furthermore, $\overline{\Phi}$ does not enforce the zero rule of $\Phi$. In cases of failure on one platform, it excludes that device from the calculation, thereby preserving portability. This is observed in the SYCL implementation of *backprop*, which fails on the Tesla T4. While $\Phi$ drops to zero, $\overline{\Phi}$ attains its maximum value by considering only the available GPU.

## 4.3 Conclusions

This chapter has assessed the potential of the SYCLomatic tool for automatically translating CUDA code into SYCL using the Rodinia benchmark suite as a case study. The results reveal that the tool achieves a high migration rate without manual intervention, while the generated warnings provide guidance for code revision and refinement where required. The findings also indicate that migration success is not necessarily determined by project dimensions or syntactic complexity in the code, but rather depends on the existence of suitable translations in SYCL or the absence of incompatible routines. Additionally, the performance experiments exhibit similar execution times in most cases for both CUDA and the migrated SYCL implementations. This trend is also observed when evaluating the performance portability metrics ($\Phi$ and $\overline{\Phi}$), indicating a slight advantage for SYCL in this context.

Overall, these results suggest that automatic migration tools may offer a promising path for moving from widely used, but generally proprietary, frameworks to cross-platform models that support the development of heterogeneous applications with a higher level of abstraction. Nevertheless, it would be interesting to examine the complexity of the migrated SYCL code in terms of maintainability and extensibility, using established metrics such as the Halstead complexity measure [245]. Furthermore, comparing the performance of the migrated code with a native SYCL implementation could determine whether the base code affects performance or whether the migration tool mitigates such differences.

## Chapter 5

# General conclusions

*I don't know where I'm going from here,*
*but I promise it won't be boring.*
— David Bowie, *New York, 1997*

As introduced in Chapter 2, computing systems at all scales have increasingly moved towards hardware heterogeneity to meet the high and specific demands of current applications. Furthermore, the wide range of accelerators with diverse characteristics has led to the emergence of cross-platform programming models, such as SYCL, which provide abstractions that simplify the programming of heterogeneous solutions. In this context, this work has explored the development of portable heterogeneous applications with Intel oneAPI, Intel's implementation of SYCL, evaluating their performance and portability on distributed HPC systems.

Chapter 3 presented the first of the studies addressing this issue. A microbenchmark was developed using oneAPI, comprising two iterative problems, one computationally intensive and the other memory-bound, to analyse the performance of CPU+GPU and CPU+FPGA configurations. Moreover, dynamic load balancing between the host and the device was applied to assess its impact on the performance. The results revealed that memory sharing with the CPU directly affected the iGPU, degrading the performance in memory-bound scenarios. In computationally intensive problems, its parallel architecture provided an advantage over the CPU, and the heterogeneous configuration delivered even better results than the iGPU by efficiently distributing the workload. Regarding the dGPUs, they achieved high performance for both problem types due to their computational power and dedicated memory. However, the CPU+dGPU configuration only outperformed the dGPU when the CPU contribution offset the communication overhead introduced by rebalancing the workload. Concerning the FPGAs and their heterogeneous configurations, they produced

non-competitive results, mainly due to the reuse of a kernel not optimised for their architecture. In addition, it was observed that input data not based on powers of two exposed performance penalties caused by processing data misaligned with the data bus width, resulting in significant communication overhead. It was also worth noting that applying dynamic load balancing at runtime proved essential for adapting the distribution of work to the evolving behaviour of the devices across iterations. This strategy was beneficial in mitigating performance drops and related inefficiencies.

The second study, explained in Chapter 4, addressed the automatic migration of the Rodinia benchmark suite from CUDA to SYCL (oneAPI) using SYCLomatic, analysing the degree of automation achieved by the tool and comparing the performance portability of the native and the migrated code. The migration success rate was 93.24%, with most warnings related to differences between the APIs, performance issues, and error handling routines. The statistical analysis did not reveal a clear relationship between the success of the migration, or the number of warnings generated, and the program dimensions, measured in number of CUDA lines, or syntactic complexity. Instead, the determining factors appeared to be more closely related to the existence of SYCL equivalents or the absence of unsupported routines. In terms of performance, CUDA and SYCL exhibited similar execution times on the two NVIDIA GPUs evaluated. Performance portability was also comparable according to the two metrics considered, although SYCL showed a slight advantage.

As general conclusions, the two studies presented demonstrate the potential of tools based on cross-platform programming models to abstract the complexities underlying heterogeneous configurations. In this case, the feasibility of Intel oneAPI has been confirmed both for developing applications targeting hardware with different architectures and vendors, and for obtaining an automatic cross-platform solution from proprietary code, thereby facilitating the adoption of such models. Additionally, the studies have also highlighted the varying degrees of maturity of the accelerators employed with respect to software development. Specifically, in Chapter 3, the FPGA was constrained not only by a non-optimised kernel but also by automated data alignment and padding operations applied when the input data did not match the bus width. Undoubtedly, FPGAs have considerable potential due to their reconfigurability, enabling them to tackle irregular problems, save energy, and process data in-network. Notwithstanding, their role as HPC accelerators and their wider adoption in mainstream high-level development will depend on the evolution of their tool ecosystem, as has already occurred with other devices such as GPUs.

Future research directions include the development of device-specific FPGA kernels to maximise the performance achieved with these accelerators. For this purpose, the exploration of other tools such as Vitis HLS is considered, since as noted in Chapter 2, Intel oneAPI has been discontinued for FPGAs and the future of this initiative remains uncertain. With regard to the performance-driven dynamic workload balance employed in this work through the IHP library, a promising line is the design of energy-aware strategies and support for multiple devices. Finally, it would be valuable

to extend the study with other cross-platform tools as well as additional accelerators from different vendors. Moreover, the evaluation of real-world problems would also provide stronger evidence to validate these programming models.

# Bibliography


[1] R. Laso, J. C. Cabaleiro, F. F. Rivera, M. C. Muñiz, and J. Alvarez-Dios, "IHP: A dynamic heterogeneous parallel scheme for iterative or time-step methods—image denoising as case study", *The Journal of Supercomputing*, vol. 77, Jan. 2021. DOI: 10.1007/s11227-020-03260-8.

[2] CERN, *Large Hadron Collider*, Accessed 18 Nov 2024. [Online]. Available: https://home.cern/science/accelerators/large-hadron-collider.

[3] L. Clissa, M. Lassnig, and L. Rinaldi, "How big is Big Data? A comprehensive survey of data production, storage, and streaming in science and industry", *Frontiers in Big Data*, vol. 6, p. 1 271 639, 2023. DOI: 10.3389/fdata.2023.1271639.

[4] J. Simon, *Large Language Models*, Accessed 18 Nov 2024. [Online]. Available: https://huggingface.co/blog/large-language-models.

[5] G. E. Moore, "Cramming more components onto integrated circuits", *Electronics Magazine*, vol. 38, no. 8, pp. 114–117, 1965.

[6] M. J. Flynn, "Very high-speed computing systems", *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966. DOI: 10.1109/PROC.1966.5273.

[7] *Message Passing Interface Forum: A Message-Passing Interface Standard (MPI)*, Accessed 17 Feb 2025, 1994. [Online]. Available: https://www.mpi-forum.org/docs/mpi-1.1/mpi-11.ps.

[8] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions", *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[9] O. A. R. Board, *OpenMP: Simple, portable, scalable SMP programming*, Accessed 17 Feb 2025, 1998. [Online]. Available: https://www.openmp.org/wp-content/uploads/cspec20.pdf.

[10] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2007, ISBN: 978-0-596-51480-8.

[11] J. Nickolls, "GPU parallel computing architecture and CUDA programming model", *Proceedings of IEEE Hot Chips 19 Symposium (HCS)*, pp. 1–12, 2007. DOI: 10.1109/HOTCHIPS.2007.7482491.

[12] *Argonne Leadership Computing Facility*, Accessed 17 Feb 2025. [Online]. Available: https://www.alcf.anl.gov/aurora.

[13] *Top 500*, Accessed 17 Feb 2025. [Online]. Available: https://top500.org/lists/top500/.

[14] *Green 500*, Accessed 17 Feb 2025. [Online]. Available: https://top500.org/lists/green500/.

[15] *SYCL: Khronos open standard for C++ heterogeneous parallel programming*, Accessed 12 Jan 2024. [Online]. Available: https://www.khronos.org/api/sycl.

[16] *The Khronos Group Inc*, Accessed 12 Jan 2024. [Online]. Available: https://www.khronos.org/.

[17] *Intel oneAPI*, Accessed 10 May 2025. [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/oneapi.html.

[18] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian, *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. United States: Apress Berkeley, CA, Jan. 2021. DOI: 10.1007/978-1-4842-5574-2.

[19] Intel Corporation, *Intel® Quartus® Prime Design Software*, Accessed 21 Apr 2025. [Online]. Available: https://www.intel.la/content/www/xl/es/products/details/fpga/development-tools/quartus-prime.html.

[20] *SYCLomatic official github*, Accessed 25 Apr 2025. [Online]. Available: https://github.com/oneapi-src/SYCLomatic.

[21] *Get Started with the Intel® DPC++ Compatibility Tool*, Accessed 25 Apr 2025. [Online]. Available: https://www.intel.com/content/www/us/en/docs/dpcpp-compatibility-tool/get-started-guide/2024-0/overview.html.

[22] N. S. Kim, D. Chen, J. Xiong, and W.-m. W. Hwu, "Heterogeneous computing meets near-memory acceleration and high-level synthesis in the post-moore era", *IEEE Micro*, vol. 37, no. 4, pp. 10–18, 2017. DOI: 10.1109/MM.2017.3211105.

[23] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis, "Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications", in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06, Tampa, Florida: Association for Computing Machinery, 2006, 125–es, ISBN: 0769527000. DOI: 10.1145/1188455.1188585.

[24] A. G. Shet, P. Sadayappan, D. E. Bernholdt, J. Nieplocha, and V. Tipparaju, "A framework for characterizing overlap of communication and computation in parallel applications", *Cluster Computing*, vol. 11, pp. 75–90, 2008. DOI: 10.1007/s10586-007-0046-3.

[25] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures", in *Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, the Netherlands, August 25-28, 2009. Proceedings 15*, Springer, 2009, pp. 863–874, ISBN: 978-3-642-03869-3. DOI: 10.1007/978-3-642-03869-3_80.

[26] E. Castillo, N. Jain, M. Casas, M. Moreto, M. Schulz, R. Beivide, M. Valero, and A. Bhatele, "Optimizing computation-communication overlap in asynchronous task-based programs", in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19, Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 380–391, ISBN: 9781450360791. DOI: 10.1145/3330345.3330379.

[27] M. Ferat, R. Pereira, A. Roussel, P. Carribault, L.-A. Steffenel, and T. Gautier, "Enhancing MPI+OpenMP task based applications for heterogeneous architectures with GPU support", in *International Workshop on OpenMP*, Springer, 2022, pp. 3–16. DOI: 10.1007/978-3-031-15922-0_1.

[28] K. Al Nuaimi, N. Mohamed, M. Al Nuaimi, and J. Al-Jaroodi, "A Survey of Load Balancing in Cloud Computing: Challenges and Algorithms", in *2012 Second Symposium on Network Cloud Computing and Applications*, IEEE, 2012. DOI: 10.1109/NCCA.2012.29.

[29] A. Cabrera, A. Acosta, F. Almeida, and V. Blanco, "A Dynamic Multi–Objective Approach for Dynamic Load Balancing in Heterogeneous Systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 10, pp. 2421–2434, 2020. DOI: 10.1109/TPDS.2020.2989869.

[30] U. Ahmed, J. C.-W. Lin, and G. Srivastava, "Heterogeneous energy-aware load balancing for industry 4.0 and IoT environments", *ACM Transactions on Management Information Systems (TMIS)*, vol. 13, no. 4, pp. 1–23, 2022. DOI: 10.1145/3543859.

[31] K. Tanaka, Y. Arikawa, T. Ito, K. Morita, N. Nemoto, F. Miura, K. Terada, J. Teramoto, and T. Sakamoto, "Communication-Efficient Distributed Deep Learning with GPU-FPGA Heterogeneous Computing", in *2020 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2020, pp. 43–46. DOI: 10.1109/HOTI51249.2020.00021.

[32] S. Zhaopeng, Z. Kuanjiu, C. Kai, and H. Shaoqi, "PCIE-Based High-Performance FPGA-GPU-CPU Heterogeneous Communication Method", in *2020 International Workshop on Electronic Communication and Artificial Intelligence (IWECAI)*, 2020, pp. 66–73. DOI: 10.1109/IWECAI50956.2020.00020.

[33] Q. Wang and D. Oswald, "Confidential Computing on Heterogeneous CPU-GPU Systems: Survey and Future Directions", *arXiv preprint*, 2024. DOI: 10.48550/arXiv.2408.11601.

[34] J. D. Foley, "Modern graphics hardware", in *Computer Graphics: Principles and Practice*. Addison-Wesley Professional, 1996, vol. 12110, ch. 38.

[35] J. H. Clark, "The geometry engine: A VLSI geometry system for graphics", *acm siggraph Computer Graphics*, vol. 16, no. 3, pp. 127–133, 1982. DOI: 10.1145/800064.801272.

[36] T. Oguchi, M. Higuchi, T. Uno, M. Kamaya, and M. Suzuki, "A single-chip graphic display controller", in *1981 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, IEEE, vol. 24, 1981, pp. 170–171. DOI: 10.1109/ISSCC.1981.1156160.

[37] I. Watanabe and S. Takagi, "Technological trajectory analysis of patent citation networks: Examining the technological evolution of computer graphic processing systems", *The Review of Socionetwork Strategies*, vol. 15, pp. 1–25, 2021. DOI: 10.1007/s12626-020-00066-1.

[38] K. Akeley, "Reality Engine graphics", in *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '93, Anaheim, CA: Association for Computing Machinery, 1993, pp. 109–116, ISBN: 0897916018. DOI: 10.1145/166117.166131.

[39] J. L. Mitchell, "RADEON 9700 shading", *State of the Art in Hardware Shading, Course Note*, vol. 17, 2002. [Online]. Available: https://www.pixelmaven.com/jason/articles/ATI/ATIHardwareShading_2002_Chapter3-1.pdf.

[40] C. McClanahan, "History and evolution of GPU architecture", *A Survey Paper*, vol. 9, pp. 1–7, 2010. [Online]. Available: https://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf.

[41] W. J. Dally, S. W. Keckler, and D. B. Kirk, "Evolution of the Graphics Processing Unit (GPU)", *IEEE Micro*, vol. 41, no. 6, pp. 42–51, 2021. DOI: 10.1109/MM.2021.3113475.

[42] J. Neider and T. Davis, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1*, 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1993, ISBN: 0201632748.

[43] *cuBLAS library*, Accessed 24 Feb 2025. [Online]. Available: https://github.com/NVIDIA/CUDALibrarySamples/tree/master/cuBLAS.

[44] *Large Scale Visual Recognition Challenge (ILSVRC) 2012*, Accessed 24 Feb 2025. [Online]. Available: https://www.image-net.org/challenges/LSVRC/2012/.

[45] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12, Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105. DOI: 10.1145/3065386.

[46] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient primitives for Deep Learning", *arXiv preprint*, 2014. DOI: 10.48550/arXiv.1410.0759.

[47] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning", in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16, Savannah, GA, USA: USENIX Association, 2016, pp. 265–283, ISBN: 9781931971331. DOI: 10.48550/arXiv.1605.08695.

[48] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library", in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[49] D. B. Kirk and W. H. Wen-Mei, "Chapter 1: Introduction", in *Programming massively parallel processors: A hands-on approach*, 3rd ed., Morgan kaufmann, 2017, pp. 1–16. DOI: 10.1016/C2020-0-02969-5.

[50] AMD, *HIP: Heterogeneous-computing Interface for Portability*, Accessed 21 Apr 2025, 2025. [Online]. Available: https://rocm.docs.amd.com/projects/HIP/.

[51] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems", *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66–73, 2010. DOI: 10.1109/MCSE.2010.69.

[52] Apple, *Metal Overview: Apple Developer*, Acessed 25 Apr 2025. [Online]. Available: https://developer.apple.com/metal/.

[53] AMD, *Hipother official Github*, Acessed 27 Apr 2025. [Online]. Available: https://github.com/ROCm/hipother.

[54] NVIDIA, *Introducing Triton: Open-source GPU programming for neural networks*, Acessed 27 Apr 2025. [Online]. Available: https://openai.com/index/triton/.

[55] NVIDIA, *NVIDIA CUDA Compiler (nvcc) documentation*, Acessed 11 Mar 2025. [Online]. Available: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[56] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software", *ACM Comput. Surv.*, vol. 34, no. 2, pp. 171–210, 2002, ISSN: 0360-0300. DOI: 10.1145/508352.508353.

[57] G. Estrin, B. Bussell, R. Turn, and J. Bibb, "Parallel Processing in a Restructurable Computer System", *IEEE Transactions on Electronic Computers*, vol. EC-12, no. 6, pp. 747–755, 1963. DOI: 10.1109/PGEC.1963.263558.

[58] A. Boutros and V. Betz, "FPGA Architecture: Principles and Progression", *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 4–29, 2021. DOI: 10.1109/MCAS.2021.3071607.

[59] W. Carter, "A user programmable reconfigurable gate array", in *Proc. Custom Integrated Circuits Conf., May 1986*, 1986. [Online]. Available: https://api.semanticscholar.org/CorpusID:60420567.

[60] S. M. S. Trimberger, "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology", *IEEE Solid-State Circuits Magazine*, vol. 10, no. 2, pp. 16–29, 2018. DOI: 10.1109/MSSC.2018.2822862.

[61] S. Gandhare and B. Karthikeyan, "Survey on FPGA Architecture and Recent Applications", in *2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN)*, 2019, pp. 1–4. DOI: 10.1109/ViTECoN.2019.8899550.

[62] A. Corporation, *Stratix ii Device Handbook*, Accessed 3 Apr 2025. [Online]. Available: https://cdrdv2-public.intel.com/654261/stratix2_handbook.pdf.

[63] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel programming for FPGAs", *arXiv preprint*, 2018. DOI: 10.48550/arXiv.1805.03648.

[64] E. D. Sozzo, D. Conficconi, A. Zeni, M. Salaris, D. Sciuto, and M. D. Santambrogio, "Pushing the Level of Abstraction of Digital System Design: A Survey on How to Program FPGAs", *ACM Comput. Surv.*, vol. 55, no. 5, 2022, ISSN: 0360-0300. DOI: 10.1145/3532989.

[65] C. Mead and L. Conway, *Introduction to VLSI Systems*. USA: Addison-Wesley Longman Publishing Co., Inc., 1979, ISBN: 0201043580.

[66] "IEEE Standard for VHDL Language Reference Manual", *IEEE Std 1076-2019*, pp. 1–673, 2019. DOI: 10.1109/IEEESTD.2019.8938196.

[67] D. Thomas and P. Moorby, *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.

[68] S. Sutherland, S. Davidmann, and P. Flake, *SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Springer Science & Business Media, 2006.

[69] Siemens EDA, *ModelSim Simulator*, Accessed 21 Apr 2025, 2024. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/modelsim/.

[70] AMD (Xilinx), *Vivado Design Suite*, Accessed 21 Apr 2025, 2024. [Online]. Available: https://www.amd.com/es/products/software/adaptive-socs-and-fpgas/vivado.html.

[71] M. Fingeroff, "Making the case for High-Level Synthesis", in *High-level synthesis: blue book*, Xlibris Corporation, 2010, ch. 1. [Online]. Available: https://cse.usf.edu/~haozheng/teach/cda4253/doc/hls/hls_bluebook_uv.pdf.

[72] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011. DOI: 10.1109/TCAD.2011.2110592.

[73] M. Fingeroff, "High-level C++ Synthesis", in *High-level synthesis: blue book*, Xlibris Corporation, 2010, ch. 1. [Online]. Available: https://cse.usf.edu/~haozheng/teach/cda4253/doc/hls/hls_bluebook_uv.pdf.

[74] AMD, *Vitis HLS*, Accessed 24 Apr 2025, 2025. [Online]. Available: https://www.amd.com/es/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html.

[75] AMD, *Vitis High-Level Synthesis User Guide (UG1399): 2024.2*, Accessed 24 Apr 2025, 2025. [Online]. Available: https://docs.amd.com/r/en-US/ug1399-vitis-hls/Introduction.

[76] *AMD Vitis™ Unified Software Platform*, Accessed 24 Apr 2025. [Online]. Available: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis.html.

[77] *FPGA Support Package for Intel® oneAPI DPC++/C++ Compiler*, Accessed 24 Apr 2025. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/fpga.html.

[78] *Intel sells off majority stake in its FPGA business*, Accessed 24 Apr 2025. [Online]. Available: https://www.networkworld.com/article/3964054/intel-sells-off-majority-stake-in-its-fpga-business.html#:~:text=In%20deal%20with%20Silver%20Lake,field-programmable%20gate%20array%20products&text=Intel%20has%20spun%20off%20its,hefty%20loss%20on%20this%20deal.

[79] *Catapult High-Level Synthesis and Verification*, Accessed 24 Apr 2025. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/.

[80] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems", in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11, Monterey, CA, USA: Association for Computing Machinery, 2011, pp. 33–36, ISBN: 9781450305549. DOI: 10.1145/1950413.1950423.

[81] *Catapult High-Level Synthesis and Verification*, Accessed 24 Apr 2025. [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html.

[82] *Catapult High-Level Synthesis and Verification*, Accessed 24 Apr 2025. [Online]. Available: https://google.github.io/xls/.

[83] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications", in *2013 23rd International Conference on Field programmable Logic and Applications*, 2013, pp. 1–4. DOI: 10.1109/FPL.2013.6645550.

[84] *Dynamatic official Github*, Accessed 24 Apr 2025. [Online]. Available: https://github.com/EPFL-LAP/dynamatic.

[85] L. Josipović, A. Guerrieri, and P. Ienne, "Invited tutorial: Dynamatic: From c/c++ to dynamically scheduled circuits", in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Seaside, CA, USA: Association for Computing Machinery, 2020, pp. 1–10, ISBN: 9781450370998. DOI: 10.1145/3373087.3375391.

[86] *Intel Announces Strategic Investment by Silver Lake in Altera*, Accessed 24 Apr 2025. [Online]. Available: https://newsroom.intel.com/corporate/intel-partner-deal-news-april2025.

[87] *PYNQ: Python productivity for adaptive computing platforms*, Accessed 24 Apr 2025. [Online]. Available: https://www.pynq.io/.

[88] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, V. Loncar, J. Ngadiuba, M. Pierini, D. Rankin, R. Rivera, S. Summers, N. Tran, and Z. Wu, "Fast Inference of Deep Neural Networks for Real-time Particle Physics Applications", FPGA '19, p. 305, 2019. DOI: 10.1145/3289602.3293986.

[89] FastML Team, *Hls4ml project*, version v1.1.0, 2025. DOI: 10.5281/zenodo.1201549.

[90] J. de Fine Licht and T. Hoefler, "hlslib: Software engineering for hardware design", *arXiv preprint*, 2019. DOI: 10.48550/arXiv.1910.04436.

[91] D. F. Bacon, R. Rabbah, and S. Shukla, "FPGA programming for the masses", *Commun. ACM*, vol. 56, no. 4, pp. 56–63, 2013, ISSN: 0001-0782. DOI: 10.1145/2436256.2436271.

[92] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are We There Yet? A Study on the State of High-Level Synthesis", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, 2019. DOI: 10.1109/TCAD.2018.2834439.

[93] C. Bobda, J. M. Mbongue, P. Chow, M. Ewais, N. Tarafdar, J. C. Vega, K. Eguro, D. Koch, S. Handagala, M. Leeser, M. Herbordt, H. Shahzad, P. Hofste, B. Ringlein, J. Szefer, A. Sanaullah, and R. Tessier, "The Future of FPGA Acceleration in Datacenters and the Cloud", *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 3, 2022, ISSN: 1936-7406. DOI: 10.1145/3506713.

[94] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming", *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998. DOI: 10.1109/99.660313.

[95] L. Sommer, J. Korinth, and A. Koch, "OpenMP device offloading to FPGA accelerators", in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2017, pp. 201–205. DOI: 10.1109/ASAP.2017.7995280.

[96] P. H. Rosso, L. Petrica, N. J. Lisa, M. Pereira, S. Rigo, H. Yviquel, V. Bonato, E. Francesquini, and G. Araujo, "Integrating Multi-FPGA Acceleration to OpenMP Distributed Computing", in *Advancing OpenMP for Future Accelerators*, Springer Nature Switzerland, 2024, pp. 49–63, ISBN: 978-3-031-72567-8. DOI: 10.1007/978-3-031-72567-8_4.

[97] H. Yviquel, M. Pereira, E. Francesquini, G. Valarini, G. Leite, P. Rosso, R. Ceccato, C. Cusihualpa, V. Dias, S. Rigo, A. Souza, and G. Araujo, "The OpenMP Cluster Programming Model", in *Workshop Proceedings of the 51st International Conference on Parallel Processing*, ser. ICPP Workshops '22, Bordeaux, France: Association for Computing Machinery, 2023, ISBN: 9781450394451. DOI: 10.1145/3547276.3548444.

[98] *OpenACC official webpage*, Accessed 27 Apr 2025. [Online]. Available: https://www.openacc.org/.

[99] *OpenACC official Github*, Accessed 27 Apr 2025. [Online]. Available: https://github.com/OpenACC.

[100] *HIP official github*, Accessed 27 Apr 2025. [Online]. Available: https://github.com/ROCm/hip.

[101] *SYCL 2020 specification*, Accessed 12 Jan 2024. [Online]. Available: https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html.

[102] *ComputeCpp Professional Edition*, Accessed 27 Apr 2025. [Online]. Available: https://developer.codeplay.com/products/computecpp/pe/home/.

[103] *Codeplay Software Ltd*, Accessed 27 Apr 2025. [Online]. Available: https://codeplay.com/.

[104] *Kronos Group 1.2.1 specification*, Accessed 12 Jan 2024. [Online]. Available: https://registry.khronos.org/SYCL/specs/sycl-1.2.1.pdf.

[105] *Computecpp: Platform support notes*, Accessed 27 Apr 2025. [Online]. Available: https://developer.codeplay.com/products/computecpp/ce/2.11.0/guides/platform-support-notes.

[106] *Intel® oneAPI Level Zero*, Accessed 27 Apr 2025. [Online]. Available: https://www.intel.com/content/www/us/en/docs/dpcpp-cpp-compiler/developer-guide-reference/2024-0/intel-oneapi-level-zero.html.

[107] *AdaptiveCpp official Github*, Accessed 27 Apr 2025. [Online]. Available: https://github.com/AdaptiveCpp/AdaptiveCpp.

[108] A. Alpay and V. Heuveline, "SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL", in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '20, Munich, Germany: Association for Computing Machinery, 2020, ISBN: 9781450375313. DOI: 10.1145/3388333.3388658.

[109] *Kokkos official webpage*, Accessed 27 Apr 2025. [Online]. Available: https://kokkos.org/.

[110] *RAJA official Github*, Accessed 27 Apr 2025. [Online]. Available: https://github.com/LLNL/RAJA.

[111] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "RAJA: Portable Performance for Large-Scale Scientific Applications", in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 71–81. DOI: 10.1109/P3HPC49587.2019.00012.

[112] *Alpaka official Github*, Accessed 27 Apr 2025. [Online]. Available: https://github.com/alpaka-group/alpaka.

[113] E. Zenker, B. Worpitz, R. Widera, A. Huebl, G. Juckeland, A. Knüpfer, W. E. Nagel, and M. Bussmann, "Alpaka – An Abstraction Library for Parallel Kernel Acceleration", in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 631–640. DOI: 10.1109/IPDPSW.2016.50.

[114] *HPX official Github*, Accessed 27 Apr 2025. [Online]. Available: https://github.com/STEllAR-GROUP/hpx.

[115] H. Kaiser, P. Diehl, A. S. Lemoine, B. A. Lelbach, P. Amini, A. Berge, J. Biddiscombe, S. R. Brandt, N. Gupta, T. Heller, K. Huck, Z. Khatami, A. Kheirkhahan, A. Reverdell, S. Shirzad, M. Simberg, B. Wagle, W. Wei, and T. Zhang, "HPX - The C++ Standard Library for Parallelism and Concurrency", *Journal of Open Source Software*, vol. 5, no. 53, p. 2352, 2020. DOI: 10.21105/joss.02352.

[116] *HIPify official github*, Accessed 25 Apr 2025. [Online]. Available: https://github.com/ROCm/HIPIFY.

[117] G. Martinez, M. Gardner, and W.-c. Feng, "CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures", in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, 2011, pp. 300–307. DOI: 10.1109/ICPADS.2011.48.

[118] *CU2CL official github*, Accessed 24 Apr 2025. [Online]. Available: https://github.com/vtsynergy/CU2CL.

[119] *OneAPI for NVIDIA® GPUs*, Accessed 25 Apr 2025. [Online]. Available: https://developer.codeplay.com/products/oneapi/nvidia/home/.

[120] *OneAPI for AMD GPUs*, Accessed 25 Apr 2025. [Online]. Available: https://developer.codeplay.com/products/oneapi/amd/home/.

[121] *TriSYCL official github*, Accessed 24 Apr 2025. [Online]. Available: https://github.com/triSYCL/triSYCL.

[122] W. Zhu, Y. Niu, and G. R. Gao, "Performance portability on EARTH: A case study across several parallel architectures", *Cluster Computing*, vol. 10, pp. 115–126, 2007. DOI: 10.1007/s10586-007-0011-1.

[123] S. McIntosh-Smith, M. Boulton, D. Curran, and J. Price, "On the Performance Portability of Structured Grid Codes on Many-Core Computer Architectures", in *Supercomputing*, Cham: Springer International Publishing, 2014, pp. 53–75, ISBN: 978-3-319-07518-1. DOI: 10.1007/978-3-319-07518-1_4.

[124] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns", *Journal of parallel and distributed computing*, vol. 74, no. 12, pp. 3202–3216, 2014. DOI: 10.1016/j.jpdc.2014.07.003.

[125] J. Neely, "DOE centers of excellence performance portability meeting", Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2016. [Online]. Available: https://asc.llnl.gov/doe-coe-mtg-2016.

[126] J. Larkin, "Performance portability through descriptive parallelism", in *Presentation at DOE Centers of Excellence Performance Portability Meeting*, 2016. [Online]. Available: https://asc.llnl.gov/sites/asc/files/2020-09/2-20_larkin.pdf.

[127] S. Pennycook, J. Sewall, and V. Lee, "Implications of a metric for performance portability", *Future Generation Computer Systems*, vol. 92, pp. 947–958, 2019, ISSN: 0167-739X. DOI: 10.1016/j.future.2017.08.007.

[128] S. J. Pennycook and J. D. Sewall, "Revisiting a Metric for Performance Portability", in *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2021, pp. 1–9. DOI: 10.1109/P3HPC54578.2021.00004.

[129] A. Marowka, "Toward a Better Performance Portability Metric", in *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2021, pp. 181–184. DOI: 10.1109/PDP52278.2021.00036.

[130] A. Marowka, "Reformulation of the performance portability metric", *Software: Practice and Experience*, vol. 52, no. 1, pp. 154–171, 2022. DOI: 10.1002/spe.3002.

[131] A. Marowka, "A comparison of two performance portability metrics", *Concurrency and Computation: Practice and Experience*, vol. 35, no. 25, e7868, 2023. DOI: 10.1002/cpe.7868.

[132] A. Marowka, "Portability efficiency approach for calculating performance portability", *Future Generation Computer Systems*, vol. 170, p. 107 826, 2025, ISSN: 0167-739X. DOI: 10.1016/j.future.2025.107826.

[133] P. J. Fleming and J. J. Wallace, "How not to lie with statistics: the correct way to summarize benchmark results", *Commun. ACM*, vol. 29, no. 3, pp. 218–221, 1986, ISSN: 0001-0782. DOI: 10.1145/5666.5673.

[134] W. F. Godoy, P. Valero-Lara, T. E. Dettling, C. Trefftz, I. Jorquera, T. Sheehy, R. G. Miller, M. Gonzalez-Tallada, J. S. Vetter, and V. Churavy, "Evaluating performance and portability of high-level programming models: Julia, Python/Numba, and Kokkos on exascale nodes", in *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2023, pp. 373–382. DOI: 10.1109/IPDPSW59300.2023.00068.

[135] R. A. Bartolomeu, R. Halver, J. H. Meinke, and G. Sutmann, "Effect of implementations of the N-body problem on the performance and portability across GPU vendors", *Future Generation Computer Systems*, vol. 169, p. 107 802, 2025, ISSN: 0167-739X. DOI: 10.1016/j.future.2025.107802.

[136] A. S. Tanenbaum and M. Van Steen, *Distributed Systems: Principles and Paradigms*, 2nd ed. Prentice Hall, 2007, ISBN: 0-13-239227-5.

[137] G. R. Andrews, *Foundations of Parallel and Distributed Programming*, 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1999, ISBN: 0201357526.

[138] P. A. Bernstein, "Middleware: A model for distributed system services", *Commun. ACM*, vol. 39, no. 2, pp. 86–98, 1996, ISSN: 0001-0782. DOI: 10.1145/230798.230809.

[139] B. C. Neuman, "Scale in distributed systems", *Readings in distributed computing systems*, 1994.

[140] H. Hussain, S. U. R. Malik, A. Hameed, S. U. Khan, G. Bickler, N. Min-Allah, M. B. Qureshi, L. Zhang, W. Yongji, N. Ghani, *et al.*, "A survey on resource allocation in high performance distributed computing systems", *Parallel Computing*, vol. 39, no. 11, pp. 709–736, 2013. DOI: 10.1016/j.parco.2013.09.009.

[141] S. Mittal and J. S. Vetter, "A survey of CPU-GPU heterogeneous computing techniques", *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, pp. 1–35, 2015. DOI: 10.1145/2788396.

[142] R. Buyya, *High Performance Cluster Computing: Architectures and Systems*. USA: Prentice Hall PTR, 1999, ISBN: 0130137847.

[143] K. O'Neal and P. Brisk, "Predictive Modeling for CPU, GPU, and FPGA Performance and Power Consumption: A Survey", in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2018, pp. 763–768. DOI: 10.1109/ISVLSI.2018.00143.

[144] K. H. Tsoi and W. Luk, "Axel: a heterogeneous cluster with FPGAs and GPUs", in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '10, Monterey, California, USA: Association for Computing Machinery, 2010, pp. 115–124, ISBN: 9781605589114. DOI: 10.1145/1723112.1723134.

[145] R. Kiełbik, K. Hałagan, W. Zatorski, J. Jung, J. Ulański, A. Napieralski, K. Rudnicki, P. Amrozik, G. Jabłoński, D. Stożek, P. Polanowski, Z. Mudza, J. Kupis, and P. Panek, "ARUZ — Large-scale, massively parallel FPGA-based analyzer of real complex systems", *Computer Physics Communications*, vol. 232, pp. 22–34, 2018, ISSN: 0010-4655. DOI: 10.1016/j.cpc.2018.06.010.

[146] W. F. Samayoa, M. L. Crespo, A. Cicuttin, and S. Carrato, "A Survey on FPGA-Based Heterogeneous Clusters Architectures", *IEEE Access*, vol. 11, pp. 67 679–67 706, 2023. DOI: 10.1109/ACCESS.2023.3288431.

[147] *Heterogeneous Accelerated Compute Clusters*, Accessed 27 Apr 2025. [Online]. Available: https://www.amd-haccs.io/.

[148] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management", in *Workshop on job scheduling strategies for parallel processing*, Springer, 2003, pp. 44–60. DOI: 10.1007/10968987_3.

[149] R. L. Henderson, "Job scheduling under the portable batch system", in *Workshop on Job Scheduling Strategies for Parallel Processing*, Springer, 1995, pp. 279–294. DOI: 10.1007/3-540-60153-8_34.

[150] G. Staples, "TORQUE resource manager", New York, NY, USA: Association for Computing Machinery, 2006, ISBN: 0769527000. DOI: 10.1145/1188455.1188464.

[151] W. Gentzsch, "Sun Grid Engine: towards creating a compute power grid", in *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2001, pp. 35–36. DOI: 10.1109/CCGRID.2001.923173.

[152] *SLURM Workload Manager: Documentation*, Accessed 20 Jul 2025. [Online]. Available: https://slurm.schedmd.com/documentation.html.

[153] J. Fang, C. Huang, T. Tang, and Z. Wang, "Parallel programming models for heterogeneous many-cores: a comprehensive survey", *CCF Transactions on High Performance Computing*, vol. 2, no. 4, pp. 382–400, 2020. DOI: 10.1007/s42514-020-00039-4.

[154] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes", in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009, pp. 427–436. DOI: 10.1109/PDP.2009.43.

[155] M. Baity-Jesi, R. Banos, A. Cruz, L. Fernandez, J. Gil-Narvion, A. Gordillo-Guerrero, M. Guidetti, D. Iniguez, A. Maiorano, F. Mantovani, *et al.*, "An FPGA-Based Supercomputer for Statistical Physics: The Weird Case of Janus", in *High-Performance Computing Using FPGAs*, Springer, 2013, pp. 481–506. DOI: 10.1007/978-1-4614-1791-0_16.

[156] T. Boku, N. Fujita, R. Kobayashi, and O. Tatebe, "Cygnus - World First Multihybrid Accelerated Cluster with GPU and FPGA Coupling", in *Workshop Proceedings of the 51st International Conference on Parallel Processing*, ser. ICPP Workshops '22, Bordeaux, France: Association for Computing Machinery, 2023, ISBN: 9781450394451. DOI: 10.1145/3547276.3548629.

[157] K. Sano, A. Koshiba, T. Miyajima, and T. Ueno, "ESSPER: Elastic and Scalable FPGA-Cluster System for High-Performance Reconfigurable Computing with Supercomputer Fugaku", in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPCAsia '23, Singapore, Singapore: Association for Computing Machinery, 2023, pp. 140–150, ISBN: 9781450398053. DOI: 10.1145/3578178.3579341.

[158] *Fugaku - RIKEN Center for Computational Science*, Accessed 20 Jul 2025. [Online]. Available: https://www.r-ccs.riken.jp/en/fugaku/.

[159] P. Schwan *et al.*, "Lustre: Building a file system for 1000-node clusters", in *Proceedings of the 2003 Linux symposium*, vol. 2003, 2003, pp. 380–386. [Online]. Available: https://www.landley.net/kdocs/mirror/ols2003.pdf#page=380.

[160] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey and Benchmarking of Machine Learning Accelerators", in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–9. DOI: 10.1109/HPEC.2019.8916327.

[161] *Frontier - Oak Ridge Leadership Computing Facility*, Accessed 20 Jul 2025. [Online]. Available: https://www.olcf.ornl.gov/frontier/.

[162] *LUMI - CSC (IT Center for Science)*, Accessed 20 Jul 2025. [Online]. Available: https://lumi-supercomputer.eu/.

[163] *MareNostrum 5 - BSC-CNS*, Accessed 20 Jul 2025. [Online]. Available: https://www.bsc.es/marenostrum/marenostrum-5.

[164] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger, "Economic models for resource management and scheduling in grid computing", *Concurrency and computation: practice and experience*, vol. 14, no. 13-15, pp. 1507–1542, 2002. DOI: 10.1002/cpe.690.

[165] *Worldwide LHC Computing Grid*, Accessed 20 Jul 2025. [Online]. Available: https://wlcg.web.cern.ch/.

[166] P. M. Mell and T. Grance, "SP 800-145. The NIST Definition of Cloud Computing", Gaithersburg, MD, USA, Tech. Rep., 2011.

[167] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "OpenStack: toward an open-source solution for cloud computing", *International Journal of Computer Applications*, vol. 55, no. 3, 2012. DOI: 10.5120/8738-2991.

[168] *Kubernetes: Production-Grade Container Orchestration*, Accessed 20 Jul 2025. [Online]. Available: https://kubernetes.io/.

[169] S. Mathew and J. Varia, "Overview of Amazon Web Services", *Amazon Whitepapers*, vol. 105, no. 1, p. 22, 2014. [Online]. Available: https://media.amazonwebservices.com/AWS_Overview.pdf.

[170] *Microsoft Azure Cloud Services*, Accessed 20 Jul 2025. [Online]. Available: https://azure.microsoft.com/es-es/products/cloud-services/.

[171] *Google Cloud: Cloud Computing Services*, Accessed 20 Jul 2025. [Online]. Available: https://cloud.google.com/.

[172] *Amazon Elastic Compute Cloud (EC2)*, Accessed 20 Jul 2025. [Online]. Available: https://aws.amazon.com/ec2.

[173] *Azure Virtual Machines*, Accessed 20 Jul 2025. [Online]. Available: https://azure.microsoft.com/es-es/products/virtual-machines.

[174] *Google Cloud: App Engine*, Accessed 20 Jul 2025. [Online]. Available: https://cloud.google.com/appengine.

[175] M. Hajibaba and S. Gorgin, "A Review on Modern Distributed Computing Paradigms: Cloud Computing, Jungle Computing and Fog Computing", *Journal of computing and information technology*, vol. 22, no. 2, pp. 69–84, 2014. DOI: 10.2498/cit.1002381.

[176] *Elastic Fabric Adapter - AWS*, Accessed 20 Jul 2025. [Online]. Available: https://aws.amazon.com/hpc/efa/.

[177] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure accelerated networking: SmartNICs in the public cloud", in *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'18, Renton, WA, USA: USENIX Association, 2018, pp. 51–64, ISBN: 9781931971430.

[178] Z. Wang, H. Huang, J. Zhang, F. Wu, and G. Alonso, "FpgaNIC: An FPGA-based versatile 100gb SmartNIC for GPUs", in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA: USENIX Association, 2022, pp. 967–986, ISBN: 978-1-939133-29-25. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/wang-zeke.

[179] W. Lin, Y. Shan, R. Kosta, A. Krishnamurthy, and Y. Zhang, "SuperNIC: An FPGA-Based, Cloud-Oriented SmartNIC", in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, USA: Association for Computing Machinery, 2024, pp. 130–141, ISBN: 9798400704185. DOI: 10.1145/3626202.3637564.

[180] P. Hu, S. Dhelim, H. Ning, and T. Qiu, "Survey on fog computing: architecture, key technologies, applications and open issues", *Journal of Network and Computer Applications*, vol. 98, pp. 27–42, 2017, ISSN: 1084-8045. DOI: 10.1016/j.jnca.2017.09.002.

[181] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed, "Edge computing: A survey", *Future Gener. Comput. Syst.*, vol. 97, no. C, pp. 219–235, 2019, ISSN: 0167-739X. DOI: 10.1016/j.future.2019.02.050.

[182] H. F. Atlam, R. J. Walters, and G. B. Wills, "Fog computing and the internet of things: A review", *big data and cognitive computing*, vol. 2, no. 2, p. 10, 2018. DOI: 10.3390/bdcc2020010.

[183] A. V. Dastjerdi and R. Buyya, "Fog Computing: Helping the Internet of Things Realize Its Potential", *Computer*, vol. 49, no. 8, pp. 112–116, 2016. DOI: 10.1109/MC.2016.245.

[184] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges", *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016. DOI: 10.1109/JIOT.2016.2579198.

[185] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK Benchmark: Past, present and future", *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003. DOI: 10.1002/cpe.728.

[186] *Alibaba Cloud*, Accessed 20 Jul 2025. [Online]. Available: https://www.alibabacloud.com/.

[187] *Introduction to Alibaba Cloud GPU Service*, Accessed 22 Jul 2025. [Online]. Available: https://www.alibabacloud.com/blog/599905.

[188] X. Wang, Y. Niu, F. Liu, and Z. Xu, "When FPGA Meets Cloud: A First Look at Performance", *IEEE Transactions on Cloud Computing*, vol. 10, no. 2, pp. 1344–1357, 2020. DOI: 10.1109/TCC.2020.2992548.

[189] *Intel Tiber AI Cloud*, Accessed 30 Jun 2025. [Online]. Available: https://ai.cloud.intel.com/.

[190] D. Lindsay, S. S. Gill, D. Smirnova, and P. Garraghan, "The evolution of distributed computing systems: from fundamental to new frontiers", *Computing*, vol. 103, no. 8, pp. 1859–1878, 2021. DOI: 10.1007/s00607-020-00900-y.

[191] D. Lukarski and M. Neytcheva, *On the Impact of the Heterogeneous Multicore and Many-Core Platforms on Iterative Solution Methods and Preconditioning Techniques*. United States: John Wiley & Sons, Ltd, 2014, ch. 2, pp. 11–32. DOI: 10.1002/9781118711897.ch2.

[192] S. Venkatasubramanian and R. W. Vuduc, "Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems", ser. Proceedings of the 23rd International Conference on Supercomputing (ICS), New York, NY, USA: Association for Computing Machinery, 2009, pp. 244–255, ISBN: 9781605584980. DOI: 10.1145/1542275.1542312.

[193] P. Benner, P. Ezzatti, E. S. Quintana-Orti, and A. Remon, "Using hybrid CPU-GPU platforms to accelerate the computation of the matrix sign function", ser. Euro-Par – Parallel Processing Workshops, Berlin, Heidelberg: Springer, Aug. 2009, pp. 132–139, ISBN: 978-3-642-14121-8. DOI: 10.1007/978-3-642-14122-5_17.

[194] P. Benner, P. Ezzatti, D. Kressner, E. S. Quintana-Ortí, and A. Remón, "A mixed-precision algorithm for the solution of Lyapunov equations on hybrid CPU–GPU platforms", *Parallel Computing*, vol. 37, no. 8, pp. 439–450, 2011, ISSN: 0167-8191. DOI: 10.1016/j.parco.2010.12.002.

[195] J. Agulleiro, F. Vázquez, E. Garzón, and J. Fernández, "Dynamic load scheduling on CPU-GPU for iterative tomographic reconstruction", ser. IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, 2012, pp. 603–608. DOI: 10.1109/ISPA.2012.90.

[196] K. Halbiniak, L. Szustak, T. Olas, R. Wyrzykowski, and P. Gepner, "Exploration of OpenCL Heterogeneous Programming for Porting Solidification Modeling to CPU-GPU Platforms", *Concurrency and Computation: Practice and Experience*, vol. 33, no. 4, e6011, 2021. DOI: 10.1002/cpe.6011.

[197] S. Belhaous, S. Chokri, S. Baroud, and M. Mestari, "Comparative Study of the Execution Time of Parallel Heat Equation on CPU and GPU", *Journal of Communications Software and Systems*, vol. 17, no. 4, pp. 350–357, Dec. 2021. DOI: 10.24138/jcomss-2021-0133.

[198] M. G. Sánchez, V. Vidal, and J. Bataller, "Peer Group and Fuzzy Metric to Remove Noise in Images Using Heterogeneous Computing", in *Euro-Par 2011: Parallel Processing Workshops*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 502–510, ISBN: 978-3-642-29737-3. DOI: 10.1007/978-3-642-29737-3_55.

[199] S. Sarjanoja, J. Boutellier, and J. Hannuksela, "BM3D image denoising using heterogeneous computing platforms", in *2015 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2015, pp. 1–8. DOI: 10.1109/DASIP.2015.7367257.

[200] R. LeVeque, *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems (Classics in Applied Mathematics Classics in Applied Mathematics)*. USA: Society for Industrial and Applied Mathematics, 2007, ISBN: 0898716292.

[201] B. Tang, G. Sapiro, and V. Caselles, "Diffusion of General Data on Non-Flat Manifolds via Harmonic Maps Theory: The Direction Diffusion Case", *International Journal of Computer Vision*, vol. 36, no. 2, pp. 149–161, 2000, ISSN: 1573-1405. DOI: 10.1023/A:1008152115986.

[202] *Intel Advisor*, Accessed 12 Jan 2024. [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html.

[203] *Intel V-Tune*, Accessed 12 Jan 2024. [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html.

[204] *oneAPI GPU Optimization Guide*, Accessed 20 Jun 2025. [Online]. Available: https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2025-0/overview.html.

[205] *FPGA Optimization Guide for Intel oneAPI Toolkits*, Accessed 20 Jun 2025. [Online]. Available: https://www.intel.com/content/www/us/en/docs/oneapi-fpga-add-on/optimization-guide/2023-2/overview.html.

[206] Intel Corporation, *Questa*: software Intel FPGA Edition*, Accessed 20 Jun 2025. [Online]. Available: https://www.intel.la/content/www/xl/es/software/programmable/quartus-prime/questa-edition.html.

[207] D. Constantinescu, A. Navarro, F. Corbera, J.-A. Fernández-Madrigal, and R. Asenjo, "Efficiency and productivity for decision making on low-power heterogeneous CPU+GPU SoCs", *The Journal of Supercomputing*, Jan. 2021. DOI: 10.1007/s11227-020-03257-3.

[208] W. Yong, Z. Yongfa, W. Scott, Y. Wang, X. Qing, and W. Chen, "Developing Medical Ultrasound Imaging Application across GPU, FPGA, and CPU Using OneAPI", ser. IWOCL'21, Munich, Germany: Association for Computing Machinery, 2021, ISBN: 9781450390330. DOI: 10.1145/3456669.3456680.

[209] G. Lupescu and N. Ţăpuş, "Design of hashtable for heterogeneous architectures", in *2021 23rd International Conference on Control Systems and Computer Science (CSCS)*, 2021, pp. 172–177. DOI: 10.1109/CSCS52396.2021.00035.

[210] E. Marinelli and R. Appuswamy, "XJoin: Portable, parallel hash join across diverse XPU architectures with oneAPI", in *DAMON 2021, 17th International Workshop on Data Management on New Hardware, Held with ACM SIGMOD/PODS, 21 June 2021, China (Virtual Event)*, ACM, Ed., 2021. DOI: https://doi.org/10.1145/3465998.3466012.

[211] E. Marinelli and R. Appuswamy, "OneJoin: Cross-architecture, scalable edit similarity join for DNA data storage using oneAPI", in *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*, ACM, Ed., Copenhagen, 2021.

[212] R. Nozal and J. L. Bosque, "Straightforward Heterogeneous Computing with the oneAPI Coexecutor Runtime", *Electronics*, vol. 10, no. 19, 2021, ISSN: 2079-9292. DOI: 10.3390/electronics10192386.

[213] N. N. Bavarsad, H. M. Makrani, H. Sayadi, L. Landis, S. Rafatirad, and H. Homayoun, "HosNa: A DPC++ benchmark suite for heterogeneous architectures", in *2021 IEEE 39th International Conference on Computer Design (ICCD)*, 2021, pp. 509–516. DOI: 10.1109/ICCD53106.2021.00084.

[214] R. Kashino, R. Kobayashi, N. Fujita, and T. Boku, "Multi-Hetero Acceleration by GPU and FPGA for Astrophysics Simulation on OneAPI Environment", in *International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPCAsia2022, Virtual Event, Japan: Association for Computing Machinery, 2022, pp. 84–93, ISBN: 9781450384988. DOI: 10.1145/3492805.3492817.

[215] T. Groth, S. Groppe, T. Pionteck, F. Valdiek, and M. Koppehel, "Hybrid CPU/GPU/APU accelerated query, insert, update and erase operations in hash tables with string keys", *Knowledge and Information Systems*, vol. 65, pp. 1–19, May 2023. DOI: 10.1007/s10115-023-01891-w.

[216] S. Li, J. Zhu, J. Han, Y. Peng, Z. Wang, X. Gong, G. Wang, J. Zhang, and X. Wang, "OneGraph: A cross-architecture framework for large-scale graph computing on GPUs based on oneAPI", *CCF Transactions on High Performance Computing*, 2023. DOI: 10.1007/s42514-023-00172-w.

[217] T. Balaji, B. Bhuvanesan, C. Vishnu, G. Revanth, and A. Akhter, "Benchmarking Complex Multiplication in Signal Processing Using Intel OneAPI: A Comparative Study on CPU, GPU and FPGA Performance", in *2024 International Conference on Sustainable Communication Networks and Application (ICSCNA)*, 2024, pp. 608–613. DOI: 10.1109/ICSCNA63714.2024.10864182.

[218] W. Liang, N. Fujita, R. Kobayashi, and T. Boku, "Using Intel oneAPI for multi-hybrid acceleration programming with GPU and FPGA coupling", in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops*, ser. HPCAsia '24 Workshops, Nagoya, Japan: Association for Computing Machinery, 2024, pp. 69–76, ISBN: 9798400716522. DOI: 10.1145/3636480.3637220.

[219] C. Campos, R. Asenjo, and A. Navarro, "Exploring data flow design and vectorization with oneAPI for streaming applications on CPU+ GPU", *The Journal of Supercomputing*, vol. 81, no. 2, pp. 1–30, 2025. DOI: 10.1007/s11227-024-06891-3.

[220] *Intel Xeon E-2176G Processor*, Accessed 12 Jan 2024. [Online]. Available: https://ark.intel.com/content/www/xl/es/ark/products/134860/intel-xeon-e2176g-processor-12m-cache-up-to-4-70-ghz.html.

[221] *Intel Xeon Platinum 8352Y Processor*, Accessed 20 Jun 2025. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/212284/intel-xeon-platinum-8352y-processor-48m-cache-2-20-ghz/specifications.html.

[222] *NVIDIA A100 Tensor Core GPU*, Accessed 20 Jun 2025. [Online]. Available: https://www.nvidia.com/en-us/data-center/a100/.

[223] *Intel Core i9-12900K Processor*, Accessed 20 Jun 2025. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/134599/intel-core-i912900k-processor-30m-cache-up-to-5-20-ghz/specifications.html.

[224] *GeForce RTX 3080 Family of Graphic Cards*, Accessed 20 Jun 2025. [Online]. Available: https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3080-3080ti/.

[225] *Intel Xeon Gold 6128 processor*, Accessed 12 Jan 2024. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/120482/intel-xeon-gold-6128-processor-19-25m-cache-3-40-ghz.html.

[226] *Intel FPGA Arria 10*, Accessed 12 Jan 2024. [Online]. Available: https://www.intel.la/content/www/xl/es/products/details/fpga/arria/10.html.

[227] H. R. Zohouri and S. Matsuoka, "The memory controller wall: Benchmarking the Intel FPGA SDK for OpenCL memory interface", ser. IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC), 2019, pp. 11–18. DOI: 10.1109/H2RC49586.2019.00007.

[228] S. R. Alcaraz, R. Laso, O. G. Lorenzo, D. L. Vilariño, T. F. Pena, and F. F. Rivera, "Performance evaluation of heterogeneous CPU+iGPU and CPU+FPGA schemes using Intel OneAPI on the cloud", in *International Conference Computational and Mathematical Methods in Science and Engineering (CMMSE)*, Rota (Spain), 2023.

[229] S. R. Alcaraz, R. Laso, O. G. Lorenzo, D. L. Vilariño, T. F. Pena, and F. F. Rivera, "Assessing Intel OneAPI capabilities and cloud-performance for heterogeneous computing", *The Journal of Supercomputing*, vol. 80, pp. 13 295–13 316, 2024, ISSN: 1573-0484. DOI: 10.1007/s11227-024-05958-5.

[230] S. R. Alcaraz, R. Laso, D. L. Vilariño, and F. F. Rivera, "SYCL for CPU+GPU Heterogeneous Computing: A Study on Integrated and Discrete GPUs", in *2025 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*, 2025, pp. 1–2. DOI: 10.1109/CLUSTERWorkshops65972.2025.11164210.

[231] J. Reinders *et al.*, "Chapter 13: Data parallel c++ in practice", in *Data Parallel C++: Revolutionizing High-Performance Programming*, J. Reinders *et al.*, Eds., Intel Press, 2021, ch. 13, pp. 297–322, ISBN: 978-0997041550.

[232] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing", in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[233] *SYCLomatic: A New CUDA-to-SYCL Code Migration Tool*, Accessed 25 Jul 2025. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/syclomatic-new-cuda-to-sycl-code-migration-tool.html.

[234] *SYCLomatic: Diagnostics Reference*, Accessed 29 Jul 2025. [Online]. Available: https://oneapi-src.github.io/SYCLomatic/dev_guide/reference/diagnostics-reference.html.

[235] *Rodinia benchmark suite*, Accessed: 19 April 2025. [Online]. Available: https://github.com/yuhc/gpu-rodinia.

[236] K. Asanović, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley", Tech. Rep. UCB/EECS-2006-183, 2006. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[237] M. Costanzo, E. Rucci, C. García-Sánchez, M. Naiouf, and M. Prieto-Matías, "Migrating CUDA to oneAPI: A Smith-Waterman Case Study", in *Bioinformatics and Biomedical Engineering: 9th International Work-Conference, IWBBIO 2022, Maspalomas, Gran Canaria, Spain, June 27–30, 2022, Proceedings, Part II*, Gran Canaria, Spain: Springer-Verlag, 2022, pp. 103–116, ISBN: 978-3-031-07801-9. DOI: 10.1007/978-3-031-07802-6_9.

[238] M. Costanzo, E. Rucci, C. García-Sanchez, M. Naiouf, and M. Prieto-Matías, "Assessing opportunities of SYCL for biological sequence alignment on GPU-based systems", *The Journal of Supercomputing*, vol. 80, no. 9, pp. 12 599–12 622, 2024. DOI: 10.1007/s11227-024-05907-2.

[239] M. Costanzo, "Viability study of SYCL as a unified programming model for heterogeneous systems based on GPUs in bioinformatics", *Journal of Computer Science & Technology*, vol. 24, 2024. DOI: 10.24215/16666038.24.e18.

[240] Z. Jin, "Experience of Migrating a Parallel Graph Coloring Program from CUDA to SYCL", Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States), Tech. Rep., Apr. 2022. DOI: 10.2172/1864412.

[241] Y. Faqir-Rhazoui and C. García, "Exploring the performance and portability of the k-means algorithm on SYCL across CPU and GPU architectures", *The Journal of Supercomputing*, vol. 79, no. 16, pp. 18 480–18 506, 2023. DOI: 10.1007/s11227-023-05373-2.

[242] C. Weckert, L. Solis-Vasquez, J. Oppermann, A. Koch, and O. Sinnen, "Altis-SYCL: Migrating Altis Benchmarking Suite from CUDA to SYCL for GPUs and FPGAs", in *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*,

ser. SC-W '23, Denver, CO, USA: Association for Computing Machinery, 2023, pp. 547–555, ISBN: 9798400707858. DOI: 10.1145/3624062.3624542.

[243] O. Jadhav, S. Agrawal, A. Srivastava, R. Rastogi, S. Wandhekar, V. SV, and J. Khemka, "Migration of CUDA Based Seismic Application to Cross-Platform SYCL Implementation", in *2023 IEEE 30th International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)*, 2023, pp. 57–58. DOI: 10.1109/HiPCW61695.2023.00017.

[244] G. Castaño, Y. Faqir-Rhazoui, C. García, and M. Prieto-Matías, "Evaluation of Intel's DPC++ Compatibility Tool in heterogeneous computing", *Journal of Parallel and Distributed Computing*, vol. 165, pp. 120–129, 2022, ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2022.03.017.

[245] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. USA: Elsevier Science Inc., 1977, ISBN: 0444002057.

# List of Figures

# List of Tables

# List of Listings

# Acronyms

**ACS** Accelerator-Centric Synthesis 48

**AI** Artificial Intelligence 2, 4, 10, 13, 21, 23, 30, 33, 61, 67, 68, 81

**AIACC** Alibaba Cloud AI Acceleration Engine 68, *see* AI

**AMI** Amazon Machine Image 67

**AOT** Ahead-of-Time 77, 78, 80

**API** Application Programming Interface 6, 15, 32, 35, 48, 49, 62, 98, 102–104, 109, 116

**APU** Accelerated Processing Unit 79

**ARM** Advanced RISC Machine 2, 11, 41, 67, 68, *see* RISC

**ASIC** Application Specific IC 2, 11, 38–40, *see* IC

**AST** Abstract Syntax Tree 99

**AWS** Amazon Web Services 4, 13, 62, 63, 67, 68

**BLAS** Basic Linear Algebra Subprograms 33

**BM3D** Block-Matching and Three-Dimensional 72

**BRAM** Block RAM 41, *see* RAM

**CESGA** Centro de Supercomputación de Galicia 3, 12, 27, 61, 81

**CGRA** Coarse-Grained Reconfigurable Array 39, 40

**CLB** Configurable Logic Block 40, 41

**CLI** Command-Line Interface 59

**EU** Execution Unit 157

**ExaFLOPS** Exa FLOPS 23, *see* FLOPS

**FF** Flip-Flop 40

**FLOPS** Floating-point Operations Per Second

**FPGA** Field Programmable Gate Array xiii, xiv, 2–5, 7, 10, 11, 13–15, 19, 23–30, 38–52, 54, 59, 60, 63, 67–69, 71, 77–82, 87–95, 101, 115, 116, 143, 144, 157

**FT3** Finisterrae III xiv, 3, 6, 12, 15, 27, 61, 81, 86, 109, 144, 157–159

**GCP** Google Cloud Platform 4, 13, 62, 67

**GE** Geometry Engine 32

**GFLOPS** Giga FLOPS 27, 66, 89, 90, 92, 93, 144, *see* FLOPS

**GPGPU** General-Purpose Computing on GPU 2, 10, 23, 32, 35, *see* GPU

**GPU** Graphics Processing Unit xiii, xiv, 2, 4–7, 10, 12–15, 19, 22–36, 38, 47–52, 59, 60, 63, 65, 67–69, 71, 72, 77–83, 90, 93–95, 98, 99, 101, 108–111, 113, 115, 116, 143, 144, 147, 148, 158–160, 164, 165

**HACC** Heterogeneous Accelerated Compute Clusters 59

**HBM** High Bandwidth Memory 35, 158

**HDL** Hardware Description Language 2, 3, 11, 23, 24, 26, 42–44, 94

**HIP** Heterogeneous-Compute Interface for Portability 35, 36, 49–51, 147

**HLS** High-Level Synthesis 3, 5, 11, 14, 23–26, 42, 44–47, 49, 88, 95, 116, 149

**HPC** High-Performance Computing xiii, xiv, 1–7, 9–13, 15, 22–27, 50, 52, 55, 58–60, 62–65, 67, 69, 78, 81, 86, 87, 89, 108, 115, 116, 143, 144

**I/O** Input/Output 33, 157

**IaaS** Infrastructure as a Service 4, 12, 62

**IaC** Infrastructure as Code 62

**IC** Integrated Circuit

**iGPU** Integrated GPU xiv, 5, 13, 14, 78, 79, 81–85, 89, 90, 92, 94, 115, 143, 148, 157, 163, *see* GPU

**IHP** Iterative Heterogeneous Parallelism 5, 13, 19, 80, 81, 116

**II** Initiation Interval 46

**ILSVRC** Large Scale Visual Recognition Challenge 33

**IoT** Internet of Things 4, 13, 64, 69

**IP** Internet Protocol 56

**IP** Intellectual Property 39, 47

**JIT** Just-in-Time 77

**LAN** Local-Area Network 55, 59

**LHC** Large Hadron Collider 21,

**LLC** Last Level Cache 90, 157

**LLM** Large Language Model 21

**LLNL** Lawrence Livermore National Laboratory 50

**LLVM** Low Level VM 98, *see* VM

**LUT** Look-Up Table 40

**ME** Migration Effort 102, 103

**MIMD** Multiple Instruction, Multiple Data 22

**ML** Machine Learning 33, 42, 47, 67

**MPI** Message Passing Interface 1, 9, 22, 59, 60, 63

**MW** MegaWatt 23

**NIC** Network Interface Card 63, 67–69

**NLP** Natural Language Processing 21

**NPU** Neural Processing Unit 2, 10, 23, 30, 67, 68

**NRE** Non-Recurring Engineering 38, 39

**NUMA** Non-Uniform Memory Access 157–160

**NVCC** NVIDIA CUDA Compiler 37, 103, *see* CUDA

**OMPC** OpenMP Cluster 48, *see* OpenMP

**OpenCL** Open Computing Language 35, 36, 46, 48–52, 60, 72, 78, 99

**OpenMP** Open Multi-Processing 1, 5, 9, 14, 22, 48, 50, 51, 60, 72, 79, 82, 83, 99, 101

**OS** Operating System 33

**PaaS** Platform as a Service 4, 12, 62

**PAL** Programmable Array Logic 38

**PBS** Portable Batch System 59

**PC** Personal Computer 30, 32

**PL** Programmable Logic 2, 11, 39, 41

**PLB** Programmable Logic Block 40

**PLD** Programmable Logic Device 38

**PS** Processing System 2, 11, 41

**R&D** Research and Development 61

**RAM** Random Access Memory 158–160,

**RISC** Reduced Instruction Set Computer

**RTL** Register Transfer Level 43, 44, 78

**SaaS** Software as a Service 4, 12, 62

**SDK** Software Development Kit 39, 46

**SGE** Sun Grid Engine 59

**SGI** Silicon Graphics, Inc 32

**SIMD** Single Instruction, Multiple Data 22, 33, 34, 50, 90, 94

**SIMT** Single Instruction, Multiple Thread 34, 36

**SLURM** Simple Linux Utility for Resources Management 19, 59

**SM** Streaming Multi-processor 34, 35, 159

**SoC** System on a Chip 39, 78

**SPIR** Standard Portable Intermediate Representation 77, 78

**SRAM** Static RAM 39, *see* RAM

**T&L** Transform & Lighting 32

**TBB** Intel Thread Building Blocks 22, 50, 78

**TCP** Transmission Control Protocol 56

**TFLOPS** Tera FLOPS 66, 157, *see* FLOPS

**TPU** Tensor Processing Unit 2, 10, 23, 30, 48, 67, 68

**UM** Unified Memory 35

**USM** Unified Shared Memory 5, 14, 49, 77, 82

**VHDL** Very High Speed Integrated Circuit HDL 42, *see* HDL

**VM** Virtual Machine 62

**WAN** Wide-Area Network 55, 57

**WLCG** Worldwide LHC Computing Grid 61, *see* LHC

**ZB** ZettaByte 21

# Appendix A

# Experimental hardware setups

## A.1 iGPU node on Intel DevCloud

This node is equipped with an Intel Xeon E-2176G [220] processor, a 6-core (12-thread) CPU based on the Coffee Lake architecture and manufactured using a 14 nm process. It operates with a clock speed of 3.7 GHz to 4.7 GHz and features 12 MB of LLC. Integrated into the CPU is the UHD Graphics P630 [220] with 24 Execution Unit (EU) and a maximum dynamic frequency of 1.2 GHz.

## A.2 FPGA node on Intel Devcloud

This node incorporates an Intel Xeon Gold 6128 [225], a 6-core (12-thread) processor based on Skylake architecture and fabricated using a 14 nm process. It operates with a clock speed of 3.4 GHz to 3.7 GHz. It includes 19.25 MB of LLC and supports Intel Hyper-Threading and Turbo Boost 2.0 technologies. The node also integrates an Intel Arria 10 FPGA [226], which offers up to 1.15 million logic elements, supports up to 1500 I/O pins, and delivers up to 1.6 TFLOPS of floating-point performance. Additionally, it supports high-speed interfaces such as PCIe and 10/25/40/100 Gigabit Ethernet.

## A.3 A100 node on FT3 supercomputer

This node comprises two Intel Xeon Platinum 8352Y processors (Ice Lake architecture), each with 32 physical cores and one hardware thread per core, totalling 64 logical CPUs. These are distributed across four NUMA nodes, with logical CPU ranges 0–15, 16–31, 32–47, and 48–63. The processors operate at a base frequency of

2.20 GHz, with frequency scaling between 800 MHz and 3.40 GHz. The system runs a 64-bit GNU/Linux operating system, specifically Rocky Linux 8.4 (Green Obsidian), and the kernel version 4.18.0. The node is equipped with 251 GiB of RAM and two NVIDIA A100-PCIE-40GB GPUs, Ampere architecture, each offering 40 GiB of HBM. The GPUs are connected via PCI Express and are visible under NUMA node 2, with affinity to logical CPUs 32–47. The system employs NVIDIA driver version 570.86.15 with CUDA Toolkit 12.2. At idle, each GPU draws approximately 57 W and exhibits minimal memory and compute utilisation. A detailed hierarchical representation of the node's topology is provided in Figure A.1.

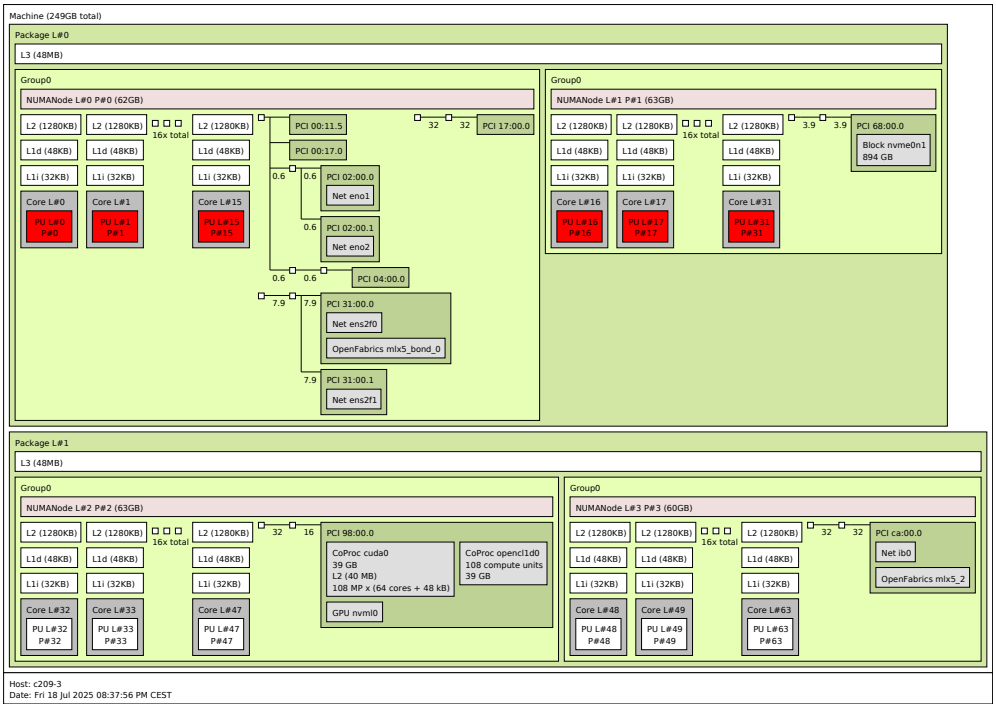



**Figure A.1:** FT3 A100 node architecture.

## A.4 Tesla T4 node on FT3 supercomputer

This node consists of two Intel Xeon Platinum 8352Y processors (Ice Lake architecture), each with 32 physical cores and one hardware thread per core, totalling 64 logical CPUs. These are distributed across four NUMA nodes, with logical CPU ranges 0–15, 16–31, 32–47, and 48–63. The processors operate at a base frequency

of 2.20 GHz, with dynamic frequency scaling between 800 MHz and 3.40 GHz. The system architecture is x86_64 with support for modern vector extensions, including AVX-512, and it is equipped with 249 GiB of RAM. The node features an NVIDIA Tesla T4 GPU, Turing architecture, which provides 16GiB of GDDR6 memory with a bandwidth of 320 GB/s. It comprises 40 SMs, each with 64 CUDA cores, totalling 2560 CUDA cores. The GPU operates at a maximum clock frequency of 1.59 GHz and supports compute capability 7.5. The installed CUDA driver and runtime versions are 11.5 and 11.2, respectively. The Tesla T4 is located on the PCI bus (ID 98:00.0) and is associated with NUMA node 2, having CPU affinity to logical CPUs 32–47. The system runs NVIDIA driver version 570.86.15, which supports CUDA version 12.8. At idle, the GPU exhibits negligible memory usage (1 MiB) and no active compute processes. A detailed hierarchical representation of the node's topology is provided in Figure A.2.

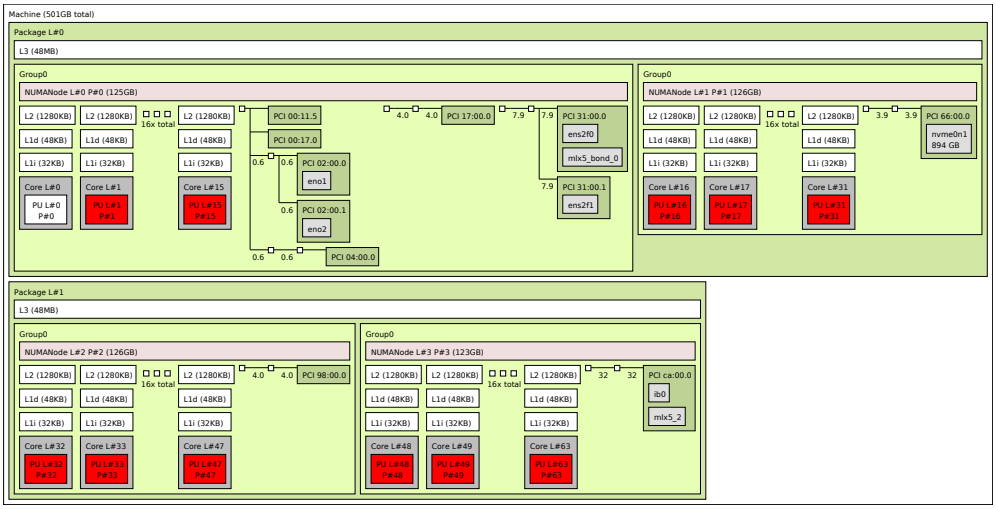



**Figure A.2:** FT3 Tesla T4 node architecture.

## A.5 CiTIUS ctheadless28 machine

This system features a single 12th Generation Intel Core i9-12900K processor with 16 physical cores and 24 logical CPUs. The architecture comprises 8 performance cores (each supporting 2 threads) and 8 efficiency cores (single-threaded), arranged under one NUMA node. The logical CPUs are enumerated from 0 to 23. The processor operates at a base frequency of 3.20 GHz, with frequency scaling between 800 MHz and 5.20 GHz. The system runs a 64-bit GNU/Linux operating system, specifically Ubuntu 22.04.2 LTS (Noble Numbat), and the kernel version 6.11.0-25-generic. The machine

has 125 GiB of RAM, with approximately 121 GiB available at idle. The memory is unified under a single NUMA node, with a total of 30 MiB of L3 cache shared across all cores. The GPU equipped is the NVIDIA GeForce RTX 3080 Ti, offering 12 GiB of GDDR6X memory. It is connected via PCIe (device 01:00.0) and is visible under the same NUMA node as the host CPU. The NVIDIA driver version is 575.51.03, and the CUDA Toolkit version used is 12.5. At idle, it consumes approximately 27 W, with minimal compute and memory utilisation. In terms of storage, the system includes two NVMe SSDs (nvme0n1 and nvme1n1) connected through separate PCI bridges, in addition to standard SATA and RAID controllers. Networking is handled by both Ethernet (enp8s0) and wireless (wlp6s0) interfaces. A detailed hierarchical representation of the machine's topology is provided in Figure A.3.

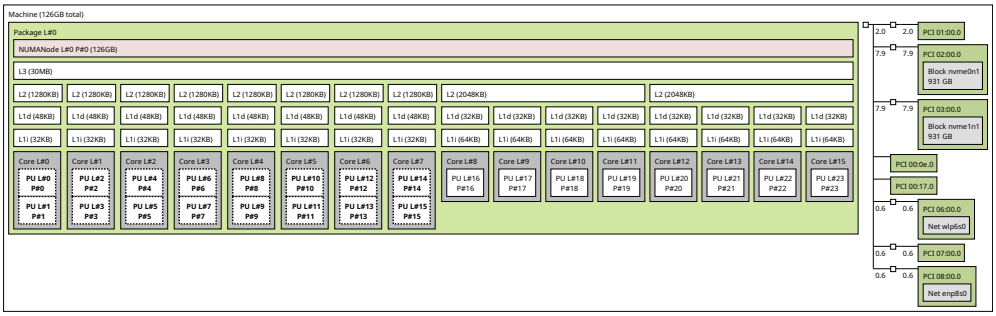



**Figure A.3:** CiTIUS *ctheadless28* machine architecture.

# Appendix B

# Additional materials

## B.1 Chapter 3

**Table B.1:** Median execution times (in seconds) for different host devices (CPUs) across two case studies, presented for input matrices of size $N \times N$.

| | Heat diffusion | | | | Image denoising | | | |
|---|---|---|---|---|---|---|---|---|
| N | Xeon-E | Xeon-Gold | Xeon-Platinum | Core-i9 | Xeon-E | Xeon-Gold | Xeon-Platinum | Core-i9 |
| 512 | 0.1953 | 0.1152 | 0.0440 | 0.0596 | 1.2286 | 0.7993 | 0.2289 | 0.1840 |
| 800 | 0.2956 | 0.2010 | 0.0490 | 0.0754 | 1.5429 | 1.8588 | 0.3349 | 0.3905 |
| 1024 | 0.2255 | 0.2731 | 0.0553 | 0.0791 | 2.5358 | 3.0409 | 1.1882 | 0.6090 |
| 1500 | 0.4908 | 0.6218 | 0.0720 | 0.2127 | 5.5654 | 6.4470 | 1.0732 | 1.1810 |
| 2048 | 1.2871 | 0.9558 | 0.0860 | 0.2284 | 10.4248 | 11.9552 | 1.9524 | 2.2509 |
| 3000 | 3.3515 | 1.9584 | 0.6193 | 1.2232 | 22.6194 | 25.4853 | 4.0888 | 4.8073 |
| 4096 | 6.3917 | 3.6391 | 2.2845 | 3.0122 | 42.2593 | 47.5052 | 7.7502 | 9.1932 |
| 6000 | 13.7172 | 7.8696 | 4.3962 | 7.0092 | 90.3080 | 102.0410 | 16.4498 | 19.0654 |
| 8192 | 25.3650 | 14.5422 | 8.1298 | 13.0572 | 169.5820 | 190.5270 | 31.1424 | 35.9953 |
| 10,000 | 37.9618 | 21.5558 | 11.9439 | 19.1208 | 250.1825 | 283.7245 | 46.3750 | 52.3559 |
| 16,384 | 102.4405 | N/A | 32.0085 | 49.9022 | 685.3385 | N/A | 134.9920 | 150.2130 |
| 20,000 | 151.2255 | N/A | 46.4855 | 73.7580 | 1011.4700 | N/A | 209.9480 | 227.1210 |

**Table B.2:** Median execution times (in seconds) for different accelerators across two case studies, presented for input matrices of size $N \times N$.

| | Heat diffusion | | | | Image denoising | | | |
|---|---|---|---|---|---|---|---|---|
| N | iGPU | A100 | RTX 3080 Ti | FPGA | iGPU | A100 | RTX 3080 Ti | FPGA |
| 512 | 0.1883 | 0.0249 | 0.0206 | 1.4452 | 0.5823 | 0.0277 | 0.0920 | 1.4851 |
| 800 | 0.3558 | 0.0302 | 0.0251 | 3.1870 | 1.3033 | 0.0362 | 0.1910 | 3.2186 |
| 1024 | 0.5457 | 0.0300 | 0.0333 | 5.4232 | 2.0907 | 0.0396 | 0.2733 | 5.5614 |
| 1500 | 1.1660 | 0.0455 | 0.0514 | 10.6434 | 4.5632 | 0.0704 | 0.5695 | 11.2492 |
| 2048 | 1.9853 | 0.0595 | 0.0729 | 21.3998 | 8.1964 | 0.0933 | 1.0090 | 21.4581 |
| 3000 | 4.8962 | 0.1194 | 0.1355 | 42.5421 | 17.3822 | 0.2348 | 1.8808 | 44.5031 |
| 4096 | 7.5936 | 0.1679 | 0.2301 | 82.6904 | 32.4657 | 0.3169 | 3.3489 | 85.3490 |
| 6000 | 16.8568 | 0.3978 | 0.4732 | 169.3560 | 69.4441 | 0.7244 | 7.1457 | 172.7500 |
| 8192 | 30.0417 | 0.6195 | 0.8063 | 319.8280 | 129.3090 | 1.2641 | 13.3566 | 341.1180 |
| 10,000 | 46.6628 | 1.0434 | 1.3079 | 469.3060 | 192.7705 | 2.0296 | 21.0584 | 482.2420 |
| 16,384 | 119.5920 | 2.4465 | 2.9888 | N/A | 518.6370 | 5.3274 | 58.0703 | N/A |
| 20,000 | 187.8775 | 3.8827 | 4.8350 | N/A | 774.3035 | 8.0852 | 89.2063 | N/A |

**Table B.3:** Median execution times (in seconds) for different heterogeneous configurations across two case studies, presented for input matrices of size $N \times N$.

| | Heat diffusion | | | | Image denoising | | | |
|---|---|---|---|---|---|---|---|---|
| N | CPU+iGPU | CPU+A100 | CPU+RTX | CPU+FPGA | CPU+iGPU | CPU+A100 | CPU+RTX | CPU+FPGA |
| 512 | 0.1321 | 0.0514 | 0.0241 | 0.2476 | 0.7159 | 0.0656 | 0.0937 | 0.9356 |
| 800 | 0.1992 | 0.0737 | 0.0299 | 1.4641 | 1.3654 | 0.0793 | 0.1773 | 4.5998 |
| 1024 | 0.2708 | 0.0742 | 0.0396 | 0.5755 | 2.2026 | 0.0844 | 0.2497 | 2.8064 |
| 1500 | 0.8173 | 0.1187 | 0.0593 | 5.0834 | 4.4386 | 0.2114 | 0.5122 | 18.0190 |
| 2048 | 1.6665 | 0.1336 | 0.0789 | 1.7439 | 7.5125 | 0.2299 | 0.9160 | 10.6059 |
| 3000 | 4.1099 | 0.1936 | 0.1620 | 16.9714 | 14.8890 | 0.2803 | 1.8580 | 48.6361 |
| 4096 | 6.7146 | 0.3608 | 0.2903 | 6.2947 | 25.6240 | 0.6919 | 3.3304 | 43.1428 |
| 6000 | 14.9160 | 0.6745 | 0.5786 | 67.7549 | 53.3036 | 1.3689 | 6.5518 | 247.8470 |
| 8192 | 23.5606 | 1.1874 | 1.0532 | 25.4222 | 96.5786 | 2.3684 | 10.9945 | 165.9615 |
| 10,000 | 39.2512 | 2.1214 | 1.6416 | 196.6840 | 142.1870 | 3.7364 | 17.1057 | 453.7185 |
| 16,384 | 92.7803 | 4.4278 | 4.1086 | N/A | 365.7045 | 9.1190 | 48.1033 | N/A |
| 20,000 | 146.0300 | 6.8534 | 6.2455 | N/A | 534.5000 | 12.0075 | 72.6597 | N/A |

**Table B.4:** Evolution of $\alpha$ for different configurations across two case studies, presented for input matrices of size $N \times N$.

| | Heat diffusion | | | | Image denoising | | | |
|---|---|---|---|---|---|---|---|---|
| N | CPU+iGPU | CPU+A100 | CPU+RTX | CPU+FPGA | CPU+iGPU | CPU+A100 | CPU+RTX | CPU+FPGA |
| 512 | 0.9433 | 0.7151 | 0.0014 | 0.9610 | 0.6119 | 0.0603 | 0.0410 | 0.7143 |
| 800 | 0.9619 | 0.6709 | 0.0025 | 0.9426 | 0.4870 | 0.0445 | 0.1063 | 0.6079 |
| 1024 | 0.9662 | 0.4559 | 0.0043 | 0.9586 | 0.5163 | 0.0307 | 0.1409 | 0.7329 |
| 1500 | 0.7093 | 0.4593 | 0.0119 | 0.9666 | 0.4478 | 0.0283 | 0.1374 | 0.5064 |
| 2048 | 0.6061 | 0.3654 | 0.0359 | 0.9635 | 0.3921 | 0.0332 | 0.1362 | 0.5991 |
| 3000 | 0.5603 | 0.3749 | 0.1177 | 0.9995 | 0.3506 | 0.0272 | 0.1763 | 0.7844 |
| 4096 | 0.4449 | 0.3098 | 0.1110 | 0.9524 | 0.2977 | 0.0182 | 0.1907 | 0.5325 |
| 6000 | 0.5048 | 0.1760 | 0.1260 | 0.9995 | 0.2861 | 0.0270 | 0.2076 | 0.5985 |
| 8192 | 0.4098 | 0.1021 | 0.1063 | 0.9284 | 0.2863 | 0.0272 | 0.2353 | 0.5290 |
| 10,000 | 0.5223 | 0.0835 | 0.0971 | 0.9995 | 0.2957 | 0.0326 | 0.2540 | 0.9995 |
| 16,384 | 0.3756 | 0.0553 | 0.0749 | N/A | 0.3148 | 0.0322 | 0.2645 | N/A |
| 20,000 | 0.4636 | 0.0581 | 0.0729 | N/A | 0.3243 | 0.0320 | 0.2651 | N/A |

**Table B.5:** Evolution of $\alpha$ and per-row execution time ($T_{\text{row}}$ in ms) across a subset of iterations for the heat diffusion problem. Results are shown for CPU-only, iGPU-only, and a CPU+iGPU configuration, with the heterogeneous case reporting the results of CPU and iGPU separately to analyse the behaviour of each device. The experiment used a 6000 × 6000 matrix.

| | Single device | | Heterogeneous configuration | | |
|---|---|---|---|---|---|
| Iteration | $T_{\text{row}}^{\text{CPU}}$ | $T_{\text{row}}^{\text{iGPU}}$ | $T_{\text{row}}^{\text{CPU}}$ | $T_{\text{row}}^{\text{iGPU}}$ | $\alpha$ |
| 1 | 0.0030 | 0.0170 | 0.0070 | 0.1360 | 0.8518 |
| 10 | 0.0020 | 0.0030 | 0.0040 | 0.0040 | 0.6241 |
| 20 | 0.0020 | 0.0030 | 0.0050 | 0.0030 | 0.4683 |
| 30 | 0.0020 | 0.0030 | 0.0050 | 0.0050 | 0.5032 |
| 40 | 0.0020 | 0.0030 | 0.0040 | 0.0040 | 0.4736 |
| 50 | 0.0020 | 0.0030 | 0.0050 | 0.0040 | 0.4812 |
| 60 | 0.0020 | 0.0030 | 0.0040 | 0.0040 | 0.4664 |
| 70 | 0.0020 | 0.0030 | 0.0040 | 0.0050 | 0.4627 |
| 80 | 0.0020 | 0.0030 | 0.0050 | 0.0040 | 0.4685 |
| 90 | 0.0020 | 0.0030 | 0.0050 | 0.0030 | 0.4648 |
| 100 | 0.0020 | 0.0030 | 0.0050 | 0.0070 | 0.4489 |

**Table B.6:** Evolution of $\alpha$ and per-row execution time ($T_{\text{row}}$ in ms) across a subset of iterations for the image denoising problem. Results are shown for CPU-only, iGPU-only, and a CPU+iGPU configuration, with the heterogeneous case reporting the results of CPU and iGPU separately to analyse the behaviour of each device. The experiment used a 6000 × 6000 matrix.

| | Single device | | Heterogeneous configuration | | |
|---|---|---|---|---|---|
| Iteration | $T_{\text{row}}^{\text{CPU}}$ | $T_{\text{row}}^{\text{iGPU}}$ | $T_{\text{row}}^{\text{CPU}}$ | $T_{\text{row}}^{\text{iGPU}}$ | $\alpha$ |
| 1 | 0.0150 | 0.0260 | 0.0320 | 0.0830 | 0.7040 |
| 10 | 0.0150 | 0.0120 | 0.0310 | 0.0110 | 0.2913 |
| 20 | 0.0150 | 0.0120 | 0.0310 | 0.0120 | 0.2745 |
| 30 | 0.0150 | 0.0120 | 0.0310 | 0.0120 | 0.2782 |
| 40 | 0.0150 | 0.0120 | 0.0320 | 0.0110 | 0.2731 |
| 50 | 0.0150 | 0.0120 | 0.0300 | 0.0120 | 0.2760 |
| 60 | 0.0150 | 0.0120 | 0.0320 | 0.0120 | 0.2732 |
| 70 | 0.0150 | 0.0120 | 0.0320 | 0.0110 | 0.2773 |
| 80 | 0.0150 | 0.0120 | 0.0300 | 0.0120 | 0.2789 |
| 90 | 0.0150 | 0.0120 | 0.0330 | 0.0120 | 0.2703 |
| 100 | 0.0150 | 0.0120 | 0.0300 | 0.0120 | 0.2754 |

## B.2 Chapter 4

To improve the interpretability of the data tables below, it should be noted that in terms of execution times, the "Sum" column represents the sum of the expressly measured parts of the code, in this case: initialisation time, memory allocations, transmission times from the host to the device and vice versa, kernel execution and memory deallocation. The "Total" column represents the entire duration of the program from start to finish, including the time spent on other routines not expressly measured. The times in this column are those used to obtain efficiencies for calculating the performance portability metrics.

**Table B.7:** Median execution times (ms) for Rodinia benchmarks using CUDA on an A100 GPU.

| Benchmark | Init | MemAlloc | HtoD | Exec | DtoH | Close | Total | Sum |
|---|---|---|---|---|---|---|---|---|
| b+tree | 0.0140 | 0.3600 | 1.9170 | 1.6605 | 0.0210 | 0.2355 | 4.5195 | 4.2080 |
| backprop | 37.4530 | 0.4698 | 19.2450 | 1.8139 | 10.2126 | 0.9360 | 640.5495 | 70.1303 |
| bfs | 0.0000 | 4.0253 | 3.9947 | 1.9550 | 0.6260 | 0.2955 | 123.0065 | 10.8965 |
| cfd | 169.7070 | 633.9260 | 2.0370 | 323.4805 | 631.9920 | 0.3835 | 2069.0500 | 1761.5260 |
| dwt2d | 269.3010 | 5.7000 | 0.0000 | 1.9570 | 0.5310 | 2.5525 | 295.3805 | 280.0415 |
| gaussian | 122.0475 | 0.2085 | 0.9715 | 66.5420 | 1.0895 | 0.1990 | 185.3470 | 191.0580 |
| heartwall | 0.0000 | 2.9020 | 19.3660 | 286.8995 | 0.0550 | 3.1215 | 951.0160 | 312.3441 |
| hotspot | 0.0000 | 0.1810 | 0.2705 | 1.5785 | 0.5890 | 0.1510 | 486.6515 | 2.7700 |
| hotspot3D | 0.0000 | 0.2245 | 1.7040 | 6.1005 | 2.6155 | 0.2140 | 4912.8113 | 10.8585 |
| huffman | 270.1265 | 0.2695 | 0.2695 | 0.0134 | 0.1775 | 0.2460 | 462.9455 | 271.1024 |
| hybridsort | 0.0000 | 357.3980 | 5.6450 | 16.5385 | 31.6085 | 1.2400 | 448.5400 | 412.4300 |
| kmeans | 62.2230 | 213.0085 | 6.8285 | 2.1040 | 0.5685 | 0.3095 | 1062.2230 | 285.0420 |
| lavaMD | 370.7510 | 0.2265 | 0.9515 | 3.9970 | 0.4370 | 0.2040 | 376.7185 | 376.5670 |
| leukocyte | 0.0004 | 2.1905 | 0.7755 | 16.4700 | 0.9730 | 0.5655 | 435.1144 | 20.9749 |
| lud | 0.2100 | 267.4765 | 0.0515 | 1.7445 | 0.4650 | 0.0880 | 270.0565 | 270.0355 |
| myocyte | 0.0000 | 0.3768 | 0.0294 | 0.1076 | 0.0554 | 0.0494 | 578.4690 | 0.6186 |
| nn | 385.8700 | 0.0980 | 0.0570 | 1.5085 | 0.1140 | 0.0910 | 401.9095 | 387.7385 |
| nw | 0.0000 | 0.1785 | 3.3555 | 2.1590 | 5.7820 | 0.1790 | 413.0115 | 11.6540 |
| particlefilter_float | 0.0000 | 0.1560 | 0.0710 | 5.1435 | 0.0350 | 0.1130 | 287.2400 | 5.5185 |
| particlefilter_naive | 0.0000 | 0.1165 | 0.3765 | 2.2845 | 0.1650 | 0.1025 | 287.6040 | 3.0450 |
| pathfinder | 206.2480 | 0.1905 | 3.8640 | 1.6665 | 0.2355 | 0.1735 | 354.9295 | 212.3780 |
| srad_v1 | 0.0000 | 253.3845 | 0.0325 | 6.8730 | 4.0855 | 0.3730 | 290.6005 | 264.7485 |
| srad_v2 | 0.0000 | 377.0135 | 2.9125 | 1.5990 | 4.4425 | 0.4165 | 505.8005 | 386.3840 |
| streamcluster | 0.0000 | 119.6653 | 330.7781 | 117.2849 | 260.5536 | 1199.4581 | 4916.5435 | 2027.7400 |

**Table B.8:** Median execution times (ms) for Rodinia benchmarks using SYCL on an A100 GPU.

| Benchmark | Init | MemAlloc | HtoD | Exec | DtoH | Close | Total | Sum |
|---|---|---|---|---|---|---|---|---|
| b+tree | 0.0150 | 1.0150 | 1.9270 | 1.6450 | 0.0250 | 0.2420 | 6.0325 | 4.8690 |
| backprop | 31.1731 | 0.6544 | 19.7786 | 2.9559 | 10.1279 | 0.9124 | 67.9564 | 65.6025 |
| bfs | 0.0000 | 1.5750 | 3.8345 | 2.5970 | 0.6630 | 0.3425 | 126.8215 | 9.0120 |
| cfd | 2.1270 | 648.8480 | 3.1715 | 221.0915 | 643.5175 | 0.3620 | 1795.2700 | 1519.1175 |
| dwt2d | 389.3865 | 6.2655 | 0.0000 | 2.6740 | 0.5580 | 2.5220 | 416.4260 | 401.4060 |
| gaussian | 129.7460 | 0.2585 | 0.8635 | 71.8635 | 1.1130 | 0.2100 | 74.4955 | 204.0545 |
| heartwall | 0.0000 | 3.2230 | 18.7560 | 416.5715 | 0.0680 | 3.6305 | 941.5710 | 442.2490 |
| hotspot | 0.0000 | 0.2305 | 0.3600 | 1.8240 | 0.5515 | 0.4370 | 659.6940 | 3.4030 |
| hotspot3D | 0.0000 | 0.2440 | 1.7750 | 3.5750 | 2.5705 | 0.2300 | 2686.7280 | 8.3945 |
| huffman | 276.3490 | 0.2825 | 0.3460 | 0.0224 | 0.2265 | 0.2670 | 288.7030 | 277.4934 |
| hybridsort | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| kmeans | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| lavaMD | 272.4490 | 0.2425 | 1.0195 | 3.8905 | 0.4420 | 0.2160 | 278.2880 | 278.2595 |
| leukocyte | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| lud | 161.3975 | 0.1150 | 0.0685 | 2.5485 | 0.2550 | 0.1015 | 164.3080 | 164.4860 |
| myocyte | 0.0000 | 0.0001 | 0.0368 | 0.0917 | 0.0670 | 0.0001 | 203.7760 | 0.1956 |
| nn | 388.5850 | 0.1140 | 0.0840 | 1.9145 | 0.1180 | 0.1010 | 402.8440 | 390.9165 |
| nw | 0.0000 | 0.2345 | 3.1000 | 3.6060 | 4.8940 | 0.2115 | 19.1100 | 12.0460 |
| particlefilter_float | 0.0000 | 0.1950 | 0.0870 | 4.2825 | 0.0455 | 0.1380 | 270.0625 | 4.7480 |
| particlefilter_naive | 0.0000 | 0.1340 | 0.5070 | 2.5110 | 0.2170 | 0.1150 | 276.7935 | 3.4840 |
| pathfinder | 206.3355 | 0.1275 | 3.8580 | 2.0630 | 0.2535 | 0.1595 | 236.6310 | 212.7970 |
| srad_v1 | 0.0000 | 271.5675 | 0.0610 | 9.9535 | 3.6685 | 0.3980 | 414.5465 | 285.6485 |
| srad_v2 | 0.0000 | 379.8615 | 3.1190 | 1.8435 | 4.3460 | 0.4460 | 484.4355 | 389.6160 |
| streamcluster | 0.0000 | 526.3337 | 440.2787 | 161.8463 | 327.5282 | 674.6060 | 4714.2739 | 2130.5929 |

**Table B.9:** Median execution times (ms) for Rodinia benchmarks using CUDA on a Tesla T4 GPU.

| Benchmark | Init | MemAlloc | HtoD | Exec | DtoH | Close | Total | Sum |
|---|---|---|---|---|---|---|---|---|
| b+tree | 0.0140 | 0.3370 | 2.0635 | 1.6720 | 0.0220 | 0.8425 | 4.9605 | 4.9510 |
| backprop | 38.6720 | 0.4801 | 20.3259 | 2.5471 | 13.7002 | 4.2206 | 278.8487 | 79.9458 |
| bfs | 0.0000 | 3.9966 | 4.3857 | 4.5430 | 0.6605 | 1.4220 | 122.7425 | 15.0078 |
| cfd | 159.3850 | 632.6910 | 1.9255 | 1354.8050 | 630.9020 | 1.0860 | 3072.4700 | 2780.7945 |
| dwt2d | 143.2790 | 5.4930 | 0.0000 | 1.8840 | 1.0050 | 2.8645 | 169.8145 | 154.5255 |
| gaussian | 121.5470 | 0.2120 | 1.0565 | 83.3615 | 1.1645 | 0.3640 | 192.3650 | 207.7055 |
| heartwall | 0.0000 | 2.8925 | 25.2705 | 516.0994 | 0.0500 | 3.5485 | 1177.5354 | 547.8609 |
| hotspot | 0.0000 | 0.2365 | 0.2790 | 1.6100 | 0.6180 | 0.2075 | 393.2870 | 2.9510 |
| hotspot3D | 0.0000 | 0.2290 | 1.8115 | 35.2180 | 2.6520 | 0.8775 | 5001.0500 | 40.7880 |
| huffman | 142.4390 | 0.2730 | 0.3100 | 0.0297 | 0.2265 | 0.3300 | 226.1825 | 143.6081 |
| hybridsort | 0.0000 | 154.6130 | 6.6300 | 62.0895 | 33.6780 | 4.7715 | 296.1060 | 261.7820 |
| kmeans | 67.8350 | 291.6400 | 9.1110 | 4.2360 | 0.6500 | 2.3280 | 1067.7155 | 375.8000 |
| lavaMD | 140.5810 | 0.2230 | 1.0520 | 88.9735 | 0.4715 | 0.3565 | 231.8535 | 231.6575 |
| leukocyte | 0.0001 | 2.1240 | 0.9455 | 53.0150 | 1.1740 | 0.6030 | 237.5086 | 57.8616 |
| lud | 0.2205 | 146.0540 | 0.0510 | 1.6545 | 0.3990 | 0.0915 | 148.4775 | 148.4705 |
| myocyte | 0.0000 | 0.1449 | 0.0246 | 0.2900 | 0.0473 | 0.0001 | 515.0280 | 0.5068 |
| nn | 143.0135 | 0.0925 | 0.0555 | 1.3555 | 0.1075 | 0.0970 | 158.7725 | 144.7215 |
| nw | 0.0000 | 0.1685 | 3.4980 | 2.0990 | 5.5495 | 0.5815 | 191.7765 | 11.8965 |
| particlefilter_float | 0.0000 | 0.1425 | 0.0720 | 25.5570 | 0.0345 | 0.1180 | 180.6675 | 25.9240 |
| particlefilter_naive | 0.0000 | 0.1095 | 0.3185 | 2.3455 | 0.1445 | 0.1085 | 157.7785 | 3.0265 |
| pathfinder | 206.7525 | 0.1885 | 4.0260 | 1.6685 | 0.2430 | 1.0425 | 318.4370 | 213.9210 |
| srad_v1 | 0.0000 | 143.3320 | 0.0330 | 13.1270 | 10.5630 | 0.4660 | 188.0140 | 167.5210 |
| srad_v2 | 0.0000 | 145.9950 | 3.0940 | 1.5730 | 6.5040 | 2.2235 | 278.9980 | 159.3895 |
| streamcluster | 0.0000 | 41.0299 | 470.5966 | 506.2511 | 330.8934 | 54.1029 | 4198.6331 | 1402.8739 |

**Table B.10:** Median execution times (ms) for Rodinia benchmarks using SYCL on a Tesla T4 GPU.

| Benchmark | Init | MemAlloc | HtoD | Exec | DtoH | Close | Total | Sum |
|---|---|---|---|---|---|---|---|---|
| b+tree | 0.0170 | 0.3525 | 2.0955 | 1.7715 | 0.0260 | 0.8770 | 5.1715 | 5.1395 |
| backprop | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| bfs | 0.0000 | 1.5690 | 4.2465 | 5.1030 | 0.6940 | 1.4100 | 126.2825 | 13.0225 |
| cfd | 2.0025 | 649.5075 | 2.3625 | 1182.0800 | 647.0345 | 0.9490 | 2757.2100 | 2483.9360 |
| dwt2d | 140.6330 | 5.8790 | 0.0000 | 2.7055 | 1.0260 | 2.9285 | 166.8120 | 153.1720 |
| gaussian | 130.1990 | 0.2580 | 0.9855 | 86.2175 | 1.1945 | 0.3655 | 89.0125 | 219.2200 |
| heartwall | 0.0000 | 3.2245 | 23.7535 | 539.5974 | 0.0620 | 3.8855 | 1043.4260 | 570.5229 |
| hotspot | 0.0000 | 0.2180 | 0.4490 | 1.6575 | 0.6240 | 0.1900 | 355.6130 | 3.1385 |
| hotspot3D | 0.0000 | 0.2525 | 1.8360 | 12.5410 | 2.6750 | 0.8965 | 2746.3850 | 18.2010 |
| huffman | 139.7400 | 0.2785 | 0.4280 | 0.0356 | 0.2300 | 0.3385 | 152.3230 | 141.0506 |
| hybridsort | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| kmeans | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| lavaMD | 141.9445 | 0.2395 | 1.1565 | 77.3990 | 0.4775 | 0.3645 | 221.6945 | 221.5815 |
| leukocyte | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| lud | 137.8825 | 0.1110 | 0.0750 | 2.5555 | 0.1925 | 0.1050 | 140.8360 | 140.9215 |
| myocyte | 0.0000 | 0.0001 | 0.0321 | 0.1450 | 0.0583 | 0.0001 | 243.4070 | 0.2356 |
| nn | 142.1300 | 0.1100 | 0.0980 | 1.8360 | 0.1255 | 0.1065 | 156.1665 | 144.4060 |
| nw | 0.0000 | 0.2280 | 3.2600 | 3.5475 | 4.4660 | 0.6090 | 19.2315 | 12.1105 |
| particlefilter_float | 0.0000 | 0.1850 | 0.0825 | 5.2985 | 0.0415 | 0.1410 | 152.3425 | 5.7485 |
| particlefilter_naive | 0.0000 | 0.1265 | 0.4300 | 2.4440 | 0.1900 | 0.1225 | 153.0525 | 3.3130 |
| pathfinder | 207.0775 | 0.1095 | 3.9760 | 2.1420 | 0.2745 | 0.1595 | 213.9135 | 213.7390 |
| srad_v1 | 0.0000 | 143.3565 | 0.0570 | 17.0165 | 10.7960 | 0.4900 | 192.0905 | 171.7160 |
| srad_v2 | 0.0000 | 142.5515 | 3.3015 | 1.8440 | 6.6165 | 2.2310 | 250.1460 | 156.5445 |
| streamcluster | 0.0000 | 61.5266 | 540.6165 | 541.8256 | 403.5385 | 74.9289 | 3591.4329 | 1622.4360 |

**Table B.11:** Data for calculating the performance portability metrics for CUDA.

| Benchmark | $e^{app}_{A100}$ | $e^{app}_{T4}$ | $\Phi$ | $\overline{\Phi}$ |
|---|---|---|---|---|
| b+tree | 1.0000 | 0.9111 | 0.9535 | 0.9555 |
| backprop | 0.4353 | 1.0000 | 0.6066 | 0.7177 |
| bfs | 0.9979 | 1.0000 | 0.9989 | 0.9989 |
| cfd | 1.0000 | 0.6734 | 0.8048 | 0.8367 |
| dwt2d | 0.5749 | 1.0000 | 0.7301 | 0.7875 |
| gaussian | 1.0000 | 0.9635 | 0.9814 | 0.9818 |
| heartwall | 1.0000 | 0.8076 | 0.8936 | 0.9038 |
| hotspot | 0.8081 | 1.0000 | 0.8939 | 0.9041 |
| hotspot3D | 1.0000 | 0.9824 | 0.9911 | 0.9912 |
| huffman | 0.4886 | 1.0000 | 0.6564 | 0.7443 |
| hybridsort | 0.6602 | 1.0000 | 0.7953 | 0.8301 |
| kmeans | 1.0000 | 0.9949 | 0.9974 | 0.9974 |
| lavaMD | 0.6155 | 1.0000 | 0.7620 | 0.8077 |
| leukocyte | 0.5459 | 1.0000 | 0.7062 | 0.7729 |
| lud | 0.5498 | 1.0000 | 0.7095 | 0.7749 |
| myocyte | 0.8903 | 1.0000 | 0.9420 | 0.9452 |
| nn | 0.3950 | 1.0000 | 0.5664 | 0.6975 |
| nw | 0.4643 | 1.0000 | 0.6342 | 0.7322 |
| particlefilter_float | 0.6290 | 1.0000 | 0.7722 | 0.8145 |
| particlefilter_naive | 0.5486 | 1.0000 | 0.7085 | 0.7743 |
| pathfinder | 0.8972 | 1.0000 | 0.9458 | 0.9486 |
| srad_v1 | 0.6470 | 1.0000 | 0.7857 | 0.8235 |
| srad_v2 | 0.5516 | 1.0000 | 0.7110 | 0.7758 |
| streamcluster | 0.8540 | 1.0000 | 0.9212 | 0.9270 |

**Table B.12:** Data for calculating the performance portability metrics for SYCL.

| Benchmark | $e^{app}_{A100}$ | $e^{app}_{T4}$ | $\Phi$ | $\overline{\Phi}$ |
|---|---|---|---|---|
| b+tree | 0.8573 | 1.0000 | 0.9232 | 0.9286 |
| backprop | 1.0000 | 0.0000 | 0.0000 | 1.0000 |
| bfs | 0.9957 | 1.0000 | 0.9979 | 0.9979 |
| cfd | 1.0000 | 0.6511 | 0.7887 | 0.8256 |
| dwt2d | 0.4006 | 1.0000 | 0.5720 | 0.7003 |
| gaussian | 1.0000 | 0.8369 | 0.9112 | 0.9185 |
| heartwall | 1.0000 | 0.9024 | 0.9487 | 0.9512 |
| hotspot | 0.5391 | 1.0000 | 0.7005 | 0.7695 |
| hotspot3D | 1.0000 | 0.9783 | 0.9890 | 0.9891 |
| huffman | 0.5276 | 1.0000 | 0.6908 | 0.7638 |
| hybridsort | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| kmeans | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| lavaMD | 0.7966 | 1.0000 | 0.8868 | 0.8983 |
| leukocyte | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| lud | 0.8571 | 1.0000 | 0.9231 | 0.9286 |
| myocyte | 1.0000 | 0.8372 | 0.9114 | 0.9186 |
| nn | 0.3877 | 1.0000 | 0.5587 | 0.6938 |
| nw | 1.0000 | 0.9937 | 0.9968 | 0.9968 |
| particlefilter_float | 0.5641 | 1.0000 | 0.7213 | 0.7821 |
| particlefilter_naive | 0.5529 | 1.0000 | 0.7121 | 0.7765 |
| pathfinder | 0.9040 | 1.0000 | 0.9496 | 0.9520 |
| srad_v1 | 0.4634 | 1.0000 | 0.6333 | 0.7317 |
| srad_v2 | 0.5164 | 1.0000 | 0.6811 | 0.7582 |
| streamcluster | 0.7618 | 1.0000 | 0.8648 | 0.8809 |

# Appendix C

# Copyright and permissions

## C.1 Articles

- S. R. Alcaraz, R. Laso, O. G. Lorenzo, D. L. Vilariño, T. F. Pena, and F. F. Rivera, "Assessing Intel OneAPI capabilities and cloud-performance for heterogeneous computing", *The Journal of Supercomputing*, vol. 80, pp. 13 295–13 316, 2024, ISSN: 1573-0484. DOI: 10.1007/s11227-024-05958-5.

> Springer Nature Book and Journal Authors have the right to reuse the Version of Record, in whole or in part, in their own thesis. Additionally, they may reproduce and make available their thesis, including Springer Nature content, as required by their awarding academic institution. Authors must properly cite the published work in their thesis according to current citation standards and include the following acknowledgement: *'Reproduced with permission from Springer Nature'*. Copyright policy: https://www.springernature.com/gp/partners/rights-permissions-third-party-distribution.

- S. R. Alcaraz, R. Laso, D. L. Vilariño, and F. F. Rivera, "SYCL for CPU+GPU Heterogeneous Computing: A Study on Integrated and Discrete GPUs", in *2025 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*, 2025, pp. 1–2. DOI: 10.1109/CLUSTERWorkshops65972.2025.11164210.

> The IEEE does not require individuals working on a thesis to obtain a formal reuse license, as stated in https://magazines.ieeeauthorcenter.ieee.org/choose-a-publishing-agreement/avoid-infringement-upon-ieee-copyright/.
>
> In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of the University of Santiago de Compostela's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

## C.2 Figures

> Except where explicitly stated, all figures presented in this thesis are original works created by the author. Figures that are not original are clearly cited in their respective captions and are included in accordance with the terms of use specified by the rights holders. These licensing terms are indicated within the corresponding figure captions, and all necessary acknowledgements have been provided in compliance with the applicable copyright and reuse policies.



Heterogeneous hardware has been increasingly adopted at all scales of computer systems to meet the high and specific demands of current applications. Given the wide diversity of accelerators available, cross-platform programming models such as SYCL have emerged to offer a unified abstraction layer. This thesis addresses the development of portable heterogeneous applications using Intel's implementation of SYCL, Intel oneAPI, and evaluates their performance on distributed HPC systems. To this end, a microbenchmark was developed to solve iterative problems, applying dynamic workload balancing and considering CPU+GPU and CPU+FPGA configurations. Furthermore, the degree of automation provided by SYCLomatic for migrating CUDA to SYCL was examined, analysing the performance portability of native and migrated code.