UNIVERSIDAD
DE MÁLAGA

Escuela de Ingenierías Industriales

Programa de Doctorado en Ingeniería Mecatrónica

Departamento de
Arquitectura de Computadores

TESIS DOCTORAL

Optimization of massive data applications on heterogeneous architectures

José Carlos Romero Moreno

Junio 2022

Dirigida por:

Dr. Rafael Asenjo Plaza
Dr. Andrés Rodríguez Moreno

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD DE LA TESIS PRESENTADA PARA OBTENER EL TÍTULO DE DOCTOR

D./Dña JOSÉ CARLOS ROMERO MORENO
Estudiante del programa de doctorado EN INGENIERÍA MECATRÓNICA de la Universidad de Málaga, autor/a de la tesis, presentada para la obtención del título de doctor por la Universidad de Málaga, titulada: OPTIMIZATION OF MASSIVE DATA APPLICATIONS ON HETEROGENEOUS ARCHITECTURES

Realizada bajo la tutorización de RAFAEL ASENJO PLAZA y dirección de RAFAEL ASENJO PLAZA Y ANDRÉS RODRÍGUEZ MORENO (si tuviera varios directores deberá hacer constar el nombre de todos)

DECLARO QUE:

La tesis presentada es una obra original que no infringe los derechos de propiedad intelectual ni los derechos de propiedad industrial u otros, conforme al ordenamiento jurídico vigente (Real Decreto Legislativo 1/1996, de 12 de abril, por el que se aprueba el texto refundido de la Ley de Propiedad Intelectual, regularizando, aclarando y armonizando las disposiciones legales vigentes sobre la materia), modificado por la Ley 2/2019, de 1 de marzo.

Igualmente asumo, ante a la Universidad de Málaga y ante cualquier otra instancia, la responsabilidad que pudiera derivarse en caso de plagio de contenidos en la tesis presentada, conforme al ordenamiento jurídico vigente.

En Málaga, a 08 de JUNIO de 2022

ROMERO MORENO JOSE CARLOS - 76873717N Digitally signed by ROMERO MORENO JOSE CARLOS - 76873717N Date: 2022.06.08 11:17:01 +02'00' Fdo.: JOSÉ CARLOS ROMERO MORENO Doctorando/a	ASENJO PLAZA RAFAEL - 24250704X Firmado digitalmente por ASENJO PLAZA RAFAEL - 24250704X Fecha: 2022.06.08 10:54:13 +02'00' Fdo.: RAFAEL ASENJO PLAZA Tutor/a
ASENJO PLAZA RAFAEL - 24250704X Firmado digitalmente por ASENJO PLAZA RAFAEL - 24250704X Fecha: 2022.06.08 10:54:39 +02'00' Fdo.: RAFAEL ASENJO PLAZA	

RODRIGUEZ MORENO ANDRES - 33383713H

Firmado digitalmente por RODRIGUEZ MORENO ANDRES - 33383713H
Fecha: 2022.06.08 10:11:11 +02'00'

ANDRÉS RODRÍGUEZ MORENO
Director/es de tesis

Edificio Pabellón de Gobierno. Campus El Ejido. 29071
Tel.: 952 13 10 28 / 952 13 14 61 / 952 13 71 10

Dr. Rafael Asenjo Plaza.
Catedrático del Departamento de Arquitectura de Computadores.
Universidad de Málaga.

Dr. Andrés Rodríguez Moreno.
Profesor Titular del Departamento de Arquitectura de Computadores.
Universidad de Málaga.

CERTIFICAN:

Que la memoria titulada *Optimization of massive data applications on heterogeneous architectures* ha sido realizada por D. José Carlos Romero Moreno, bajo nuestra dirección en el Departamento de Arquitectura de Computadores de la Universidad de Málaga, y constituye la Tesis que presenta para optar al grado de Doctor en Ingeniería Mecatrónica.

Málaga, 8 de Junio de 2022

ASENJO PLAZA RAFAEL - 24250704X

Firmado digitalmente por ASENJO PLAZA RAFAEL - 24250704X
Fecha: 2022.06.08 10:52:38 +02'00'

Dr. Rafael Asenjo Plaza.
Tutor y codirector de la tesis.

RODRIGUEZ MORENO ANDRES - 33383713H

Firmado digitalmente por RODRIGUEZ MORENO ANDRES - 33383713H
Fecha: 2022.06.08 10:10:11 +02'00'

Dr. Andrés Rodríguez Moreno.
Codirector de la tesis.

UNIVERSIDAD
DE MÁLAGA

Autorización para la lectura de la Tesis e Informe de la utilización de las publicaciones que la avalan

Los abajo firmantes declaran, bajo su responsabilidad, que autorizan la lectura de la tesis del doctorando D.José Carlos Romero Moreno, con DNI 76873717-N titulada *Optimization of massive data applications on heterogeneous architectures* y que ninguna de las publicaciones que avalan dicha tesis ha sido utilizada en tesis anteriores.

Málaga, 8 de Junio de 2022

ASENJO PLAZA RAFAEL - 24250704X

Firmado digitalmente por ASENJO PLAZA RAFAEL - 24250704X
Fecha: 2022.06.08 10:55:38 +02'00'

Dr. Rafael Asenjo Plaza.
Tutor y codirector de la tesis.

RODRIGUEZ MORENO ANDRES - 33383713H

Firmado digitalmente por RODRIGUEZ MORENO ANDRES - 33383713H
Fecha: 2022.06.08 10:09:12 +02'00'

Dr. Andrés Rodríguez Moreno.
Codirector de la tesis.

A mi familia, amigos y compañeros

Acknowledgments

El doctorado es un viaje largo y arduo, desafiante pero a la vez gratificante, que no podría terminar con éxito sin el apoyo de muchas personas. Quiero expresar mi gratitud a todos los que han contribuido a hacer posible esta tesis.

Me gustaría comenzar agradeciendo a todos mis compañeros de laboratorio y fatigas (sin ningún orden específico): Fran, Villegas, Ricardo, Pedrero, Denisa, Herruzo, Rubén, Felipe, Paula, Jimmy, Bernabé, Andrés, Iván y aquellos que haya podido olvidar. Querría resaltar a mis compañeros de promoción Andrés e Iván. Los tres empezamos con mucha ilusión este camino tan duro y con muy poca idea de dónde nos metíamos, ayudándonos desde el día uno, con esas interminables reuniones para solucionar el mundo en la "sala comanche", con la ilusión siempre presente de que lo queríamos conseguir. Dentro de poco podremos decir que lo conseguimos, que todos somos ya doctores.

From my period in Edinburgh, I would like to thank all the nice people that I met in the EPCC, starting by my Host Murray Cole, my room mates Pablo, Alberto y Elena, and specially to Rosa, who welcomed me as a friend from day one and taught me so much during my stay.

Mi más profundo agradecimiento a mis coautores y directores de tesis. Todos ellos forman un equipo excepcional con perfiles únicos. Antonio Vilches, co-autor, por saber transmitirme su enorme capacidad de trabajo, motivación, y su pasión por la investigación y el trabajo bien hecho. Andrés, co-director de tesis, por su mirada atenta al detalle en cada código y texto, siempre encontrando erratas, mejoras y consejos. M. Ángeles, a quien considero mi tercera co-directora, por su visión analítica y atenta al detalle, centrada siempre en el núcleo del problema a resolver y las contribuciones, le agradezco enormemente su infinita paciencia y todas las valiosas sugerencias que me ha dado a lo largo de esta tesis. Y Rafa, mi co-director, tutor y la persona que me dio la oportunidad de descubrir mi pasión por la docencia y la investigación. Nunca podré agradecer suficiente que confiaras en mi para ser tu estudiante de doctorado y me permitieras descubrir

el mundo académico, del que ahora no pienso otra cosa que no sea dedicarme a él toda la vida. Una persona con una infinita capacidad de trabajo, verdadera vocación, entusiasmo y conocimiento omnisciente desde la visión más general del problema hasta el detalle de programación más minúsculo. Siempre dispuesto a enseñar, tender una mano amiga, dedicando el tiempo que haga falta. Y más aun este último año con una situación tan adversa, siempre habéis estado ahí MA y tu. Gracias.

Agradecer a todo el personal administrativo y técnicos, que nos han hecho la vida más fácil y han aguantado todos nuestros errores y novatadas estoicamente: Carmen, Juanjo y Paco.

En el ámbito personal, a mi familia, empezando por mi mesa de tres patas: mi madre y mi hermana. No hay texto para poder describir todo lo que me habéis aportado, enseñado y querido durante toda mi vida. Sé que cada decisión que tome y cada paso que dé, ahí estaréis siempre apoyándome y queriéndome de forma incondicional. Y yo a vosotras. Gracias Lorena, por haber traído a la familia los dos mejores regalos que podríamos nunca haber deseado: mis sobrinos Jorge y Valeria. Y finalmente pero no menos importante, María. Mi compañera de viaje y mi mejor amiga. Nunca serás consciente de todo lo bueno que has traído a mi vida, lo que me aportas y enseñas cada día, lo que he crecido a tu lado y lo inmensamente feliz que me haces cada día. Gracias por haberme apoyado, querido y acompañado estos duros años de forma incondicional. Esta tesis no la habría acabado sin ti. Lo mejor está por venir, sea en París, Edimburgo o Berlín. De ahora en adelante, por fin, lo descubriremos y disfrutaremos juntos. Os quiero.

Finalmente, mencionar las fuentes de financiación que han permitido que pueda llevar a cabo esta tesis y las estancias: El proyecto del Gobierno de España TIN2016-80920-R, el proyecto HPC-EUROPA3 (INFRAIA-2016-1-730897) con el apoyo del EC Research Innovation Action bajo el programa H2020, y los programas de la Junta de Andalucía PID2019-105396RB-I00, P20-00395-R y UMA18-FEDERJA-108.

Abstract

In the last few years, the heterogeneous architectures have become dominant in each part of the computing industry: from heterogeneous CPUs able to focus on performance or efficiency, to GPU accelerators joining multi-core CPUs within the same chip, to Systems on Chip that integrate DSPs, FPGAs and many other types of processor in the same area. Hardware manufacturers have placed the responsibility for explicit accelerator management on software developers. In general, writing high performance programs for heterogeneous architectures is a complex task. Programming for this kind of platforms requires the understanding of new hardware concepts, orchestration of different parallelism levels, the explicit management of different memory spaces and synchronizations between processing units. Moreover, heterogeneous architectures suffer from performance portability, as one program can exhibit unequal performance on different devices. The main problem this thesis addresses is the optimization of real-life irregular massive data problems for heterogeneous architectures. Irregular problems are those in which the distribution of computational load is not regular and vary along the iteration space, exhibiting in most of the cases data and control divergences. This approach is achieved by providing models that, in runtime, find the best parallelization and workload balance for, not only the multi-core CPU, but also the accelerator in the heterogeneous architecture deployed. The main motivation of this thesis is the fact that there is no implementation with optimal solution for heterogeneous architectures for two massive data, real-life and complex problems widely used in big data fields: time series and the skyline problem.

In the first part of this thesis, we focus on the motifs/discord discovery problem for time series, taking as a starting point the state-of-the-art algorithm, the Matrix Profile. The workload of the Matrix Profile is an irregular problem which can be modeled and regularized in runtime to get the optimal parallelization and distribution of workload for different accelerators. We present the first heterogeneous implementations for the matrix profile computation for CPU + GPU architectures and CPU + FPGA using a High Performance FPGA with inte-

grated High Bandwidth Memory, HBM. To the best of our knowledge, we provide the first FPGA implementation of a Matrix Profile algorithm using High Level Synthesis, HLS. We experimentally evaluate three heterogeneous schedulers comparing performance and energy consumption in these architectures. We propose *Fastfit*, a hierarchical scheduler: (1) at the system level, it efficiently balances workload among the FPGA and the CPU cores; and (2) at device level, it computes an even partition of the diagonals of the Matrix Profile so that all FPGA IPs complete their assignment at the same time. We also develop a methodology based on a model to optimize the memory bandwidth usage of HBM Banks in a High Performance FPGA with integrated High Bandwidth Memory. We validate the accuracy of our models, finding that it outperforms state-of-the-art previous schedulers by achieving up to 99.4% of ideal performance.

In the second part, we tackle the problem of computing the skyline operator over a stream of independent data queries targeting a heterogeneous architecture comprised of a multi-core CPU and an integrated GPU. The skyline problem is an optimization problem which minimizes a N-Dimensional dataset into the smallest subset. The workload for each input query is highly dependent on the multidimensional distribution of points and vary with no pattern during execution. Hence, it is impossible to accurately model the workload of the dataset at runtime. This kind of irregularities offers the possibility to optimize using a fine-grained partition to keep the heterogeneous architecture busy while keeping balance between devices during all the execution. We contribute with a novel implementation, based on oneAPI, of the state-of-the-art *SkyAlign* algorithm and evaluate its performance both on GPU and on CPU. We design a graph-based engine, SkyFlow, and propose two heterogeneous approaches for skyline computation over a stream of data queries: SkyFlow-CG (Coarse-grained) and SkyFlow-FG (Fine-grained). Coarse-grained keeps two skyline computations in parallel, one per device, while in Fine-grained a single skyline computation is split between the CPU and GPU devices. We present two policies for scheduling the skyline computation of arriving data queries between devices in the Coarse-grained approach, where each device has a queue. The first strategy (Work Conserving) keeps the devices busy by offloading queries to the shortest queue. The second approach (Heterogeneous Earliest Finish Time) enqueues the incoming query on the device queue in which it will finish earlier using a model that estimates the execution time with negligible overhead. Our experimental results show that in our streaming scenarios and datasets, our heterogeneous CPU+GPU approaches always outperform previous only-CPU and only-GPU state-of-the-art implementations up to 6.86x and 5.19x, respectively, and they fall below 6% of ideal peak performance at most.

In the end, this thesis has resulted in schedulers and models to efficiently optimize two relevant massive data applications, using heterogeneous architectures. We can optimize performance, productivity or energy consumption if we select the most suitable scheduling strategy and programming model. The scientific community can apply these approaches to accelerate data processing using heterogeneous architectures in other problems with similar requirements.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

CG	Coarse-grained
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DPC++	Data Parallel C++
DPL	Data Parallel Library
DSP	Digital Signal Processor
FG	Fine-grained
FPGA	Field Programmable Gate Array
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphic Processing Unit
HBB	Heterogeneous Building Blocks
HBM	High Bandwidth Memory
HetMP	Heterogeneous Matrix Profile
HPC	High Performance Computing
IP	Intellectual Property
MP	Matrix Profile
NPU	Neural Processing Unit
oneTBB	oneAPI Threading Building Blocks
OpenCL	Open Computing Language

OpenMP	Open Multi Processing
PCM	Processor Counter Monitor
RQ	Research Question
SoC	System on a Chip
STL	Standard Template Library
SVM	Shared Virtual Memory
TBB	Threading Building Blocks
TPU	Tensor Processing Unit
USM	Unified Shared Memory

1 Introduction

Since the paradigm shift from single-core CPU to multi-core architectures, the structure of hardware architectures has been continuously progressing. Nowadays, not only the performance has become a main variable to consider in the development of new architectures but also the energy consumption. In the last few years, the heterogeneous architectures have become dominant in each part of the computing industry: from supercomputers to embedded systems or even mobile architectures. The development of task-specific devices in heterogeneous architectures (GPU, NPU, TPU, FPGA) that co-work with CPUs in SoC have allowed not only an improvement of the energy efficiency over traditional multi-core systems, but also performance-wise. In this work we discuss the challenges and analyze the strengths of using heterogeneous architectures to optimize performance and energy consumption to solve massive data problems.

In this introduction we present the transition to heterogeneous architectures from multi-core era in Section 1.1. Section 1.2 covers the challenge of programming heterogeneous architectures, as they request more effort from the programmer. The topics covered in this Section include the use of lower-level libraries to exploit the potential of the architecture, the data partitioning and the load balance among the available devices. Section 1.3 highlights the motivations of this thesis motivations and addresses the unresolved problems we are going to study. Section 1.4 states the thesis objectives and the research questions addressed. Finally, Section 1.5 presents the thesis contributions and Section 1.6 depicts the thesis structure.

1.1. The transition to heterogeneous architectures

According to Moore's law [1], over the past fifty years the number of transistors per chip, which is a short and simple way to measure their performance, has doubled every two years. This fact has been accompanied by an increase in frequency and power consumption that in the mid-2000s became unsustainable. Such limit led to stabilize the frequency in order to restrain the power consumption and to continue the improvement of technology the multi-core architecture emerged. The industry shifted its strategy for continuing performance growth from increasing the frequency of the core, to adding more cores in a chip. This innovation brought developers a challenge to take advantage of the new architectures: now they have to adapt the code with specific software to leverage the multi-core capabilities of the new chips. Until the multi-core era, the software engineers developed their codes without worrying about performance: each new iteration of architectures gave more performance to the same code. Since the dawn of the multi-core era, the burden of increasing the performance of the implementations has fallen on the programmer shoulders [2].

The last decade has seen a revolution in hardware architecture design, with chips moving from multi-core systems to heterogeneous architectures. Now the chips do not contain only replications of the same core (multi-core systems), but task-specialized devices co-working together with the CPU in a SoC. Examples of this specialized devices are:

- **GPU**. Designed initially for computer graphics and image processing. They mainly exploit massive data-level parallelism requiring less energy than CPU. Nowadays they are widely used in many fields of industry and research and are known as GPGPU (standing for General Purpose GPU) since they can accelerate general applications that have traditionally been run on (general purpose) CPU, ranging from embarrassingly parallel tasks [3] to more irregular applications like ray tracing [4].

- **FPGA**. This architecture is made of logic blocks designed to be reconfigured in a specific electronic device for implementing basic or complex logic functions [5]. This device offers very low power consumption, since it only uses the logic/transistors required to implement the specific function for which the FPGA has been reconfigured. Usually, the FPGA performance increments comes from exploiting pipeline parallelism, what makes this architecture quite suitable for stream processing.

- **DSP**. Devices specifically designed for digital signal processing such as audio signal processing, telecommunications, digital image processing, mostly found in embedded systems [6].

- **NPU and TPU**. These devices are designed specifically for machine learning processing [7]. They offer optimal performance in neural networks processing, since its architecture is designed specifically for matrix multiplications.

All of these devices offers significant performance and energy efficiency if they are used for specific task instead of CPU. However, these benefits do not come for free, since they need to be programmed by using specific libraries and programming techniques. Finding out the right way to leverage the capabilities of these new architectures is now responsibility of the programmer. The following section discusses what these programming libraries offer, their limitations and challenges.

1.2. The challenge of heterogeneous programming

Traditional parallel architectures, multi-core chips, works with the task parallel paradigm programming. Each application is divided into smaller tasks executed on the different cores available in the chip, aiming at distributing the work and improving the overall application performance. The division of the workload in smaller tasks and their distribution among available cores makes parallel programming a challenge for traditional programmers, since getting the optimal parallelization of an application can easily become a really challenging task.

Heterogeneous computing is the paradigm in which the parallelization of an application is performed in an heterogeneous architecture made of different devices. This context offers even more complexity in the parallelization, since the orchestration of tasks among devices has to take into account the capabilities of each one in order to take advantage of the architecture. The key idea of this paradigm is to assign the tasks which fit better for a particular device, optimizing the code for each device. For example, let's consider a particular code with two main parts: one with an embarrassingly parallel loop and other with a huge number of conditions and random memory access. The first one will execute faster on a GPU and the second one will fits better on a CPU. In this thesis we focus on heterogeneous CPU+GPU and CPU+FPGA architectures to solve real problems and its optimizations to leverage these architectures.

Along with the explosion of different heterogeneous architectures to solve specific problems, the number of heterogeneous frameworks and libraries has also increased. Some of them are for proprietary devices as CUDA [8] only for NVIDIA devices, while others are available for different architectures and cross-platforms such as OpenCL [9], SYCL [10] or the recently introduced oneAPI [11]. Others less common are RenderScript [12] and Mare [13] which are focused on optimizing heterogeneous applications for mobile devices.

Although these frameworks help in harnessing the heterogeneous devices by adding the accelerators along with the multi-core CPU, the challenges for leveraging heterogeneous programming has hindered the general adoption of these frameworks. These challenges open a research opportunity in which heterogeneous programming models and libraries can be proposed in order to facilitate the adoption of these heterogeneous platforms for the vast majority of developers and, hence, democratize the "heterogeneous programming".

1.3. The challenge of optimizing massive data applications on heterogeneous architectures

In heterogeneous programming, the computation is divided into tasks and offloaded into the different devices that constitute the heterogeneous platform. Optimizing an application via heterogeneous programming requires a thorough knowledge of both the application and the heterogeneous architecture on which it will be executed. Furthermore, each device has different features in terms or performance, energy consumption and parallelization capabilities. The developer has to hone his skills to find the optimal solution with the optimal resources, which becomes one of the main challenges that comes with heterogeneous programming.

The main problem this thesis addresses is the optimization of real-life irregular massive data problems for heterogeneous architectures.

Irregular problems are those in which the computational load is not regular and vary along the iteration space. In some problems the irregularity can be regularized. This approach is achieved by providing models that, at runtime, find the best parallelization and workload balance for, not only the multi-core CPU, but also the accelerator in the heterogeneous architecture deployed. On other problems, however, modeling the program before runtime is neither feasible nor possible, the workload distribution is completely unpredictable and may even change depending on previous results or input received. A possible solution in this

case is a fine-grained partitioning of the workload to keep the balance between the devices.

This issue is even more challenging and remains unsolved for many real-life problems where the complexity of the implementation of the codes makes harder to model the best parallelization or finding the optimal workload distribution.

Several approaches has been proposed in the literature to address this problem. Some of them propose an offline training phase in order to figure out the best workload partition among devices [14, 15, 16, 17]. The strategies these approaches tackle are launching small executions to profile the problem or use machine learning methods in order to build a model which predict the optimal workload partition. The drawbacks of these approaches are the ones of the offline training techniques: (1) overhead of training, particularly for large datasets and/or (2) unsuitability of training, specially for irregular applications, in which the optimal workload partition can not be estimated offline so that adaptive schedulers has to be devised to solve the partitioning problem at runtime.

Recent work has proposed solutions for optimizing irregular applications using a runtime model that is agnostic to the application being run [18, 19]. While these approaches get excellent performance in most of the codes, for specific real-life and complex irregular problems, a general optimization without knowledge of the application falls short of fully exploiting the heterogeneous architecture.

This thesis is motivated by the fact that there is no implementation with optimal solution for heterogeneous architectures for two data massive, real-life and complex problems widely used in big data fields: time series and the skyline problem. We focus on proposing an optimal solution for these two problems. On the one hand, for time series we focus in the motifs/discord discovery problem, taking as a starting point the state-of-the-art algorithm, the Matrix Profile. Previous implementations of the matrix profile [20] made an embarrassingly parallelization of the problem, at the cost of a huge computational load. State-of-the-art implementations [21, 22] optimize the sequential calculation of the matrix profile by rearranging and skipping operations, but at the cost of creating an irregular problem in its parallelization. However, the workload can be modeled and regularized at runtime to get the optimal parallelization and distribution of workload for different accelerators, as we will see in Chapters 3 and 4.

On the other hand, the skyline problem is a highly irregular problem in which the workload can not be modeled, and this fact hampers an optimal workload distribution. The skyline problem is an optimization problem widely used for multi-criteria decision making. It allows to minimize a N-Dimensional dataset into the smallest subset. In this thesis we focus on the computation of a stream

of skyline queries. The workload for each input query is highly dependent on the multidimensional distribution of points and vary with no pattern during execution. Hence, not until the complete iteration space is computed, is it impossible to accurately model the workload of the dataset. This kind of irregularities offers the possibility to optimize using a fine-grained partition to keep the heterogeneous architecture busy while keeping balance between devices during all the execution. As a starting point we take the state-of-the-art algorithms for CPU [23] and GPU [24], as we will detail in Chapter 5.

In this sense, models and scheduling strategies for these two applications are proposed to optimize performance and energy consumption on different heterogeneous architectures, such as CPU+GPU and CPU+FPGA.

1.4. Thesis Objectives and Research questions

Despite the fact that there are already different solutions proposed, as mentioned in previous section, there is still a lot of room for improvement in optimizing irregular data massive applications on heterogeneous architectures. Although these approaches fits well for regular applications, real applications with complex workflow can difficult its parallelization and optimization using generic templates, so that custom and fine-grained solutions become necessary. Also, the introduction of new accelerators such as FPGA with HBM requires specific models and schedulers to leverage its capabilities. In this thesis we aim to fill this gap in literature, providing not only the first heterogeneous implementations, but specific high level models and schedulers for our two complex and data massive applications targeted: the matrix profile and the streaming skyline computation. We use not only the traditional accelerator such as GPU, but also new specific accelerators as FPGA with integrated HBM. An HBM-capable FPGA needs a specific scheduler and model to fully leverage the architecture that it is not solved in literature yet. In the case of the skyline problem with its unpredictable workload, the general parallel solutions do not solve the heterogeneous parallelization problem, needing a custom fine-grained solution.

For this reason, the research questions(RQs) to be answered in this PhD. thesis are the following:

RQ #1: Is it possible to develop an optimal heterogeneous implementation for the state-of-the-art Time Series algorithm?

RQ #2: Can FPGA with HBM capabilities be an efficient accelerator for Time Series computation? And more specifically: Is it possible to develop a scheduler

which leverage such accelerator in an heterogeneous implementation for the state-of-the-art Time Series algorithm?

RQ #3: Is it possible to develop an optimal heterogeneous implementation for the state-of-the-art skyline algorithm and a model to optimize its performance?

RQ #4: In the context of the skyline computation, can a multi-algorithm scheduler be developed for a continuous streaming of datasets that maps the input received into the best-suited device to optimize the overall system performance?

In order to answer these RQs and taking into account that we target heterogeneous systems made of CPU+GPU and CPU+FPGA, the following objectives has been fulfilled during this PhD:

1. Analyze the state-of-the-art for time series and skyline computation, finding the gaps in the literature for heterogeneous implementation of such algorithms.
2. Understand CPU, GPU and FPGA architectures and how to adapt such algorithms to these architectures to maximize its performance.
3. Profile the algorithms, analyzing bottlenecks and different approaches for an optimal parallelization.
4. Develop an heterogeneous implementation on CPU+GPU and CPU+FPGA for time series computation and CPU+GPU for the skyline. Profile the heterogeneous implementations, to find an analytical model of its behavior and predict its execution time. Analyze impact and workload distribution among devices to fine tune the model.
5. Design performance and energy models that, at runtime, after a small profiling step, predict the workload balance with best performance on a CPU+GPU and CPU+FPGA heterogeneous architecture for such algorithms.
6. Compare our models with the different state-of-the-art approaches found in the literature to objectively measure its performance.

1.5. Thesis Contributions

This thesis is supported by several journal and conference contributions in the area of heterogeneous computing. The main contributions of this thesis which answer the previous research questions are summarized as follows:

1. In the context of time series computation:
   - We experimentally compare an ideal static distribution of the Matrix Profile computation [21, 22] with a dynamic one based on work stealing (TBB) for multi-core architectures, showing that the latter is better suited for commodity platforms as mobiles, desktops or servers.
   - We present the first heterogeneous implementation for the matrix profile computation for CPU + GPU architectures. We extend a previously developed heterogeneous parallel_for template [25] to implement heterogeneous parallel_reduce computations. We propose two alternatives for the OpenCL kernel that implements the matrix profile computation on the GPU. A precise but slower one is based on OpenCL atomic operations to guarantee the correct implementation of the reduction operations. An imprecise but faster alternative avoids the OpenCL atomic operations but results in accuracy losses.
   - We experimentally evaluate three heterogeneous schedulers (*Static*, *Dynamic* and *LogFit*) [18] that require a different input from the user. The simplest ones need the percentage of work (*Static*), or the size of the chunk of iterations (*Dynamic*), that should be offloaded to the GPU. The most elaborated one (*LogFit*) automatically computes the work granularity (and the offload ratio) required to make the most out of the available devices in the system.
   - We present, to the best of our knowledge, the first FPGA implementation of a Matrix Profile algorithm using High Level Synthesis, HLS. We tune the matrix profile algorithm [21, 22] for FPGA execution. We contribute with an efficient heterogeneous CPU + FPGA implementation that reduces the execution time and energy consumption with respect to the only-CPU approach.
   - We propose *Fastfit*, a hierarchical scheduler: (1) at the outer/inter-device/system level, it efficiently balances workload among the FPGA and the CPU cores using a strategy that calculates a near optimal partitioning of the work for each device. For it, our scheduler uses an analytical model that assumes that an FPGA IP is internally implemented as a pipeline from which it estimates the near-optimal FPGA chunk size that maximizes the device throughput; and (2) at inner/intra-device/device level, it computes an even partition of the diagonals of the Matrix Profile so that all FPGA IPs complete their assignment at the same time.
   - We develop a methodology based on a model to optimize the memory bandwidth usage of HBM Banks in a High Performance FPGA

with integrated High Bandwidth Memory. Our model allows to easily find the minimum number of active HBM banks that reduce power consumption while ensuring maximum aggregated bandwidth.

2. In the context of the skyline computation:

   - We contribute with a novel SYCL-based implementation of the *SkyAlign* algorithm and evaluate its performance both on GPU and on CPU.
   - We design a graph-based engine, SkyFlow based on oneAPI, and propose two heterogeneous approaches for skyline computation over a stream of data queries: SkyFlow-CG (Coarse-grained) and SkyFlow-FG (Fine-grained). Coarse-grained keeps two skyline computation in parallel, one per device, while in Fine-grained a single skyline computation is split between the CPU and GPU devices. We validate the suitability of each approach for different streaming scenarios.
   - We present two policies for scheduling the skyline computation of arriving data queries between devices in the Coarse-grained approach, where each device has a queue. The first strategy (Work Conserving) keeps the devices busy by offloading queries to the shortest queue. The second approach (Heterogeneous Earliest Finish Time) estimates the execution time for the arriving query on each device. To such end, we develop a model that, taking small chunks of points at runtime, estimates the execution time of an arriving query with negligible overhead. That estimated time is used to enqueue the incoming query on the device queue in which it will finish earlier.

The aforementioned contributions have been published in international conferences [26, 27] , national conference [28], and journals [29, 30] ranked by the ISI Journal Citation Report (JCR).

1.6. Thesis Structure

The rest of this thesis is structured as follows:

- Chapter 2 motivates the need to develop models to optimize applications for heterogeneous architectures. Then, an overview of the state-of-the-art in heterogeneous architectures is presented, along with the state-of-the-art for the real-life data massive applications this thesis tackles.

- Chapter 3 introduces the state-of-the-art in time series computation and the-state-of-the-art algorithm for motif/discord discovery, the matrix profile. We cover the literature of this algorithm from sequential implementation to parallel implementations and its current applications. First, we propose an heterogeneous implementation of the state-of-the-art version of the Matrix Profile, and how the synchronization and workload balancing between devices have been addressed. Later, different existing scheduling strategies are compared to optimize the heterogeneous implementation proposed.

- Chapter 4 extends the Matrix Profile computation problem introduced in Chapter 3. By the time this work was developed, new state-of-the-art was released, which was adapted to our heterogeneous implementation. This new heterogeneous implementation has been implemented in a FPGA with HBM capabilities. Previous scheduling strategies could not leverage this new accelerator architecture, so we propose in this work a novel hierarchical scheduler to optimize Matrix Profile computation in this new heterogeneous architecture.

- Chapter 5 describes the skyline problem, its complexities and different approaches in its current implementations. We propose an heterogeneous implementation for the state-of-the-art algorithm. In addition to this, a model with a scheduling strategy is proposed to optimize the workload balance for this implementation. Additionally a multi-algorithm scheduler is proposed for solving streaming skyline computations, which allows to dispatch each input dataset to the best suited device.

Experimental results are covered within the scope of each chapter to ease the readability of this thesis. The last chapter wraps up with the thesis conclusions and potential lines of future work.

2 Background and Related Work

In this chapter, we provide a background on the topics covered in this thesis: from the emergence of heterogeneous architectures, the programming models proposed to fully exploit them [31], to the challenges that data massive applications must address when ported to this type of architectures.

Nowadays there exists a huge variety of heterogeneous architectures made of CPUs and accelerators (GPU, FPGA, DSP, NPU) manufactured in a single SoC. The high performance capabilities and energy efficiency provided by these SoCs have made them pervasive in almost every computing area. They go from battery powered devices (embedded boards, smartphones, laptops) to large supercomputing systems (like the current top positions in both the general TOP500list or its Green500 version [32]).

However, these architectures come with a lot of unsolved challenges. Leveraging performance and the promised energy consumption efficiency thanks to its heterogeneous capabilities require three main skills from the programmer when porting the applications to these systems: (1) a deeper understanding of the application to that is ported; (2) a well-founded knowledge of the programming model used and its optimization functionalities, and (3) a solid knowledge of the heterogeneous architecture and the features exposed to the programming model to help optimize the new code implementation.

To help meet these new demands, also new techniques are required to facilitate finding the optimization possibilities for each application and new programming models that allow exploiting the capabilities of the architectures in a high-level and friendly way for the programmer.

In this thesis, we aim to design and apply the aforementioned optimization strategies to new heterogeneous programming models for leveraging heterogeneous architectures to accelerate specific algorithms widely used in data massive and Big Data applications. Therefore, it is necessary to understand the fundamentals covered in this section. At first, Section 2.1 covers the history of evolution from single processors to heterogeneous chips in SoC. Section 2.2 describes the details of the heterogeneous architectures used in this thesis. Section 2.3 details the programming models and the main functionalities that allow the exploitation of these architectures in order to achieve their maximum performance. Section 2.4 delves into the algorithms optimized in this thesis, covers its theoretical background, previous works and the existing challenges for porting them to heterogeneous systems. Finally, Section 2.5 presents the main related work in strategies for scheduling heterogeneous applications for both accelerators used in this thesis, GPU and FPGA, which is a key topic extensively studied in this work.

2.1. Hardware evolution

In 1965, Gordon E. Moore predicted that the number of transistors within a chip will double every two years[33], the so-called Moore's Law. Figure 2.1 shows the evolution of processors features for the last 50 years [34].

The number of transistors follows a straight line in the logarithmic y-axis, demonstrating the exponential growth Moore's Law predicted. Until the early 2000s, this increment translated into exponential performance growth. However, the performance increment collapsed after that date since performance also depended on other processor features that were stalling.

As can be seen in Figure 2.1 one of the reasons is the collapse of frequency increment. The frequency grew by three orders of magnitude, from 1MHz (in 1975) to 1GHz (in 2000). Three main factors were limiting performance growth [35]:

- **The Power wall**. It is a consequence of the non-linear relationship between frequency and power consumption increment. The increase in the number of transistors was accompanied by an increase in frequency and, therefore, performance until this increase was unsustainable in power consumption. This issue is known as the power density problem. The increment in voltage supply and frequency has stopped growing to maintain stable power density in chips. Nowadays, the frequency is kept around 3GHz.

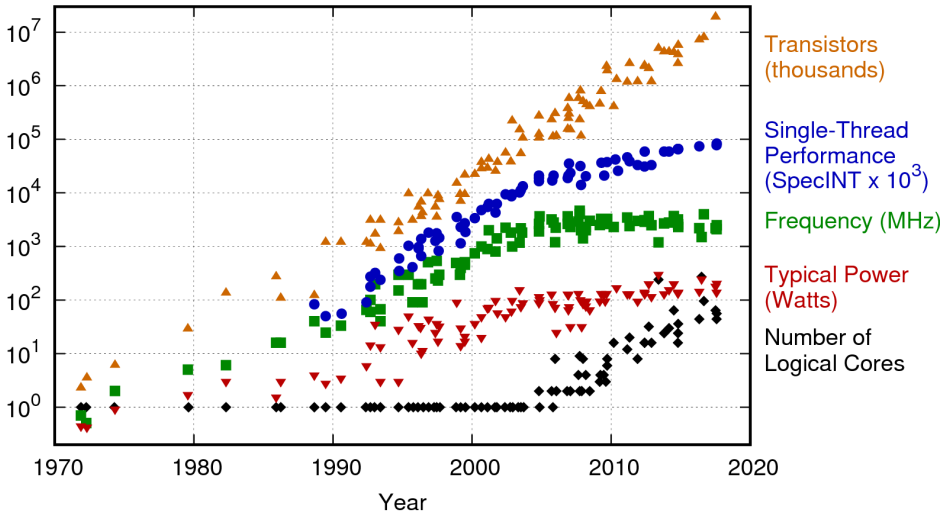


Figure 2.1: Processor features trends for the last 50 years. Data collected and plotted by Karl Rupp. Originals authors of this plot are M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten [34].

- **The Memory wall**. It appears in memory-bound codes, which have a higher number of memory operations (read and write) than arithmetic operations. The two main factors that affect memory operations are bandwidth (the rate at which data is read or stored) and latency (time between a memory request operation and it is finished). Although bandwidth still grows with new generations of chips, latency increases and becomes a limiting factor.

- **The ILP wall**. Traditional techniques like pipeline, branch predictors, and superscalar instruction issuer have extracted automatically low-level parallelism and provided increments in performance executing the same sequential code. They also have reached their limit. This effect can be seen in Figure 2.1 where performance for single-threaded runs does not scale anymore with transistors growth.

Before these factors appeared, software developers did not care about hardware in the single-core era since performance increased in every new generation of chips with no modification in their sequential codes.

These factors forced processor designers to move from single-core to multi-core and heterogeneous processors to efficiently leverage the highest number of transistors available on the chip. The trend is to increase the number of cores included in heterogeneous processors. However, hardware engineers are not only increasing the number of CPU cores but also including other specific accelerators to leverage performance in the tasks for which they have been designed. The *free lunch* to automatically increase performance in sequential codes is over. Now software engineers have to consider the architecture on which the code executes and adapt its applications to achieve performance from the heterogeneous devices. They are responsible for knowing in detail the different heterogeneous architectures available, targeting the most suitable architecture for the application, and how to extract the most of them.

2.2. Heterogeneous architectures

Heterogeneous architectures can come in two different approaches: (1) SoC where a single chip hosts the main device (CPU) and the accelerator, known as an integrated accelerator; (2) the accelerator is away and connected to the CPU by the PCI-express of the motherboard, known as a discrete accelerator. Both have their advantages and drawbacks. Integrated design allows faster communications sharing memory resources, more suitable for memory-bound problems, but is limited by bandwidth and power consumption of the entire chip. Discrete designs offer more performance and power availability but slower communication between devices. This situation creates potential bottlenecks if there are many memory operations. In this section, we present the two heterogeneous accelerators used in this work: an integrated GPU and a discrete FPGA.

2.2.1. GPU architecture

Figure 2.2 shows the architecture of a Intel processor Gen 11 SoC with an integrated GPU [36], similar to the one used in this thesis.

The architecture implements, in a SoC, a multi-core CPU processor along with an integrated GPU. A ring-based topology connects CPU cores, GPU and a shared Last Level Cache (LLC), facilitating all system memory transactions.

The GPU architecture's main area is divided into 8 sub-slices. Each sub-slice contains 8 Execution-Units (EU), making 64 EUs. The EU is the foundational building block in the GPU architecture, as can be seen in Figure 2.2. The EU com-

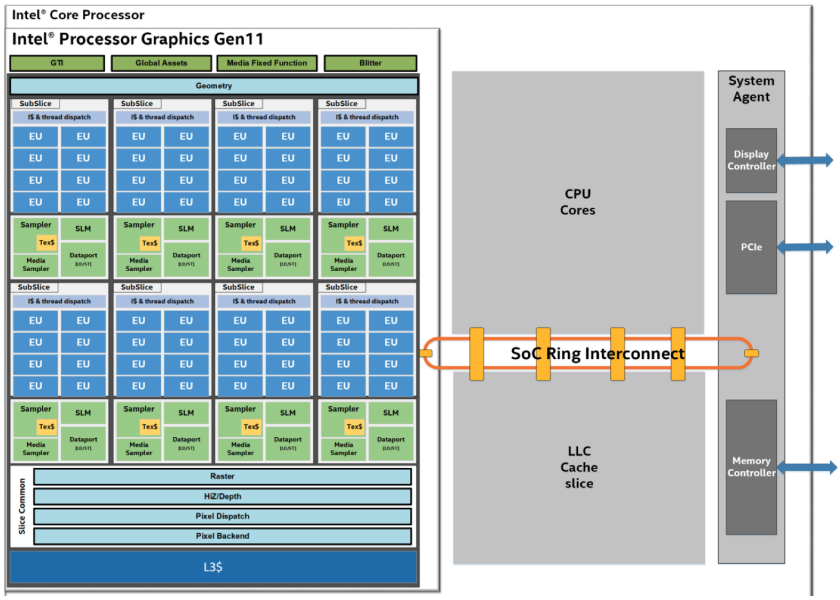


Figure 2.2: Intel Core Processor Graphics Architecture Gen 11. [36]

bines simultaneous multi-threading and fine-grained interleaved multi-threading (SMT and IMT). The EU are processors that combine multiple single instructions, multiple data arithmetic logic units, SIMD ALUS. They are distributed in a pipeline across multiple threads, allowing high throughput in integers and floating points operations. Depending on the software workload, all the threads within an EU could be executing the same kernel code or different ones. Furthermore, the fine-grained structure guarantees a flow of instructions being computed non-stop. This structure allows hiding latency for memory scatter/gather or more prolonged operations.

Let us look at the memory distribution, starting from the bottom to the top. We can see that each subslice has a shared local memory (SLM). This proximity to the EUs provides higher efficiency with low latency, increasing the effectiveness rate for atomic operations. This local memory is separated from the L3 cache shared by all subslices. Above those, a DRAM physical memory is shared between the GPU and the CPU. This unified memory architecture leverages power efficiency and programmability advantages over discrete memory transfers from host-to-device or device-to-host through PCI-Express. The main advantage of this architecture is the zero-copy buffer transfers between CPU and

GPU since the physical memory is shared. This performance is improved by adding an LLC shared with the main memory.

2.2.2. FPGA architecture

A field-programmable gate array, FPGA, is a reconfigurable integrated circuit [37]. In contrast to CPUs and GPUs having a fixed hardware structure to which a program maps, the FPGA can build custom hardware to implement a specific program. FPGA belongs to the group of customs accelerators, such as ASICs (Application Specific Integrated Circuit). On the one hand, an ASIC generally outperforms an FPGA for the specific task they are designed for but at a higher cost, requiring longer development times and more money invested. On the other hand, an FPGA is cheaper in money and development time and can be reprogrammed for each application developed.

Figure 2.3 shows an overview of the FPGA architecture. It consists of a grid of configurable logic blocks, known as adaptive logic modules (ALMs), and specialized blocks, such as digital signal processing (DSP) or random-access memory (RAM) blocks. These blocks are connected through configurable routing interconnects to implement the required digital circuits. The total number of ALMs, DSP and RAM blocks used in an implemented digital circuit is known as the *FPGA area* that the design uses.

The basic block of an FPGA, the ALM, is made of a lookup table (LUT) and a register from which the compiler can build any Boolean logic circuit. The DSP block implements specific support for fixed-point and floating-point arithmetic, which reduces the need of building such logic from the general-purpose ALMs. Finally, the RAM blocks use a high amount of memory cells to implement the memory used by the applications.

2.2.2.1. FPGA with HBM

High Bandwidth Memory (HBM) is a high-speed computer memory interface for 3D-stacked synchronous dynamic random-access memory (SDRAM). HBM achieves higher bandwidth while using less power in a substantially smaller form factor than regular RAM such as DDR4 or GDDR5. This higher bandwidth is achieved by stacking up to eight DRAM dies and connecting them to the memory controller on a CPU, GPU or FPGA.

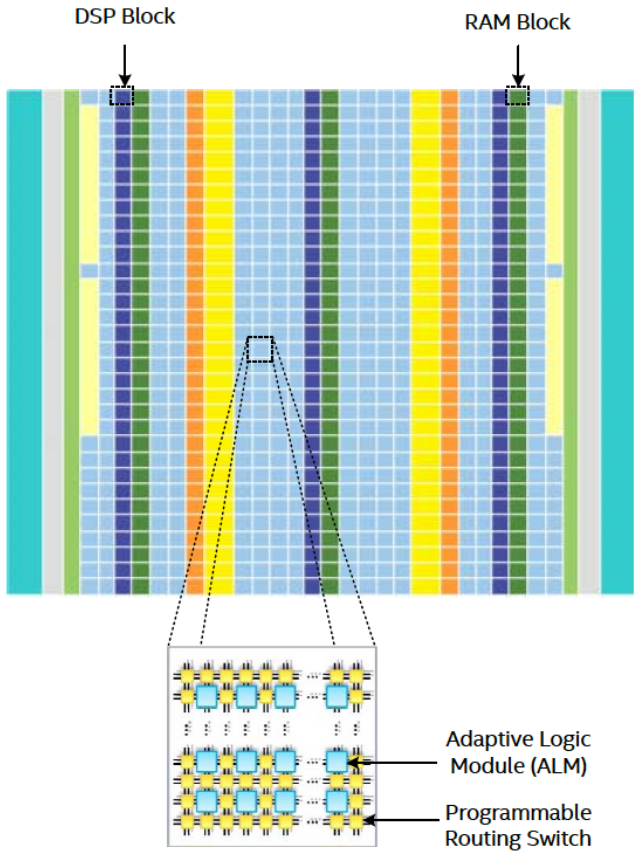


Figure 2.3: Intel FPGA architecture overview [37].

In this work, we have used an FPGA with HBM: The Intel's Stratix 10 MX2100 FPGA with integrated HBM2 memory [38]. This device has 32 pseudo HBM2 memory channels.

Each channel has 512MB, making a total of 16GB. The Intel Stratix 10 MX offers 10x more bandwidth versus current discrete memory solutions such as DDR4 SDRAM. Traditional DDR4 DIMMs provide 21 GBps bandwidth, while 1 HBM2 tile reaches up to 256 GBps. The Intel Stratix 10 MX integrates up to two HBM2 devices in a single package, enabling a maximum memory bandwidth of 512 GBps. It is designed for high-performance computing, enabling acceleration of memory-bound applications. Both traditional *Hardware design language*(HDL) and higher

abstraction C, C++, OpenCL-based and oneAPI-based tool flows are supported. It includes an optimized board support package (BSP) for the Intel OpenCL SDK and Intel oneAPI SDK. It also includes a Board Management Controller (BMC) for advanced system monitoring and control, which greatly simplifies platform integration and management. This architecture poses challenges from a programmer's point of view, detailed below.

The BSP maintains the physical partitioning between the 32 pseudo-channels (PCs). The BSP-to-kernel interface is comprised of 32 separate global memory interfaces that ultimately connect to the physically separate PCs. The PCs are partitioned because of the difficulty of creating a soft-logic in one ample address space with reasonable maximum frequency (Fmax) and area while providing maximal throughput. By partitioning the PCs, the compiler will create a small memory system for each PC that does not interact with other memory systems, resulting in better Fmax and throughput. This approach differs from previous reference BSPs where multiple DDR banks were combined to form a single homogeneous memory system. Therefore, the global memory accesses in the application must be partitioned in the same way. The application developer must explicitly specify which memory system (i.e. which PC) each global memory pointer must access in the kernel code. This assignation is done using the buffer_location attribute, as shown in Figure 2.4.

A global memory pointer without a memory system label will cause the compiler to implement by default that pointer in the first memory system (i.e. HBM0). Therefore, when porting an existing design to this architecture with no code modifications, all global memory accesses will target HBM0. For kernels needing to operate on memory buffers larger than 512MB, which is the maximum storage of each HBM2 PC, the host application must split the buffer across two or more PCs and correspondingly access them using two or more global memory pointers in the kernel. Figure 2.4 shows an example of a kernel code using the triad metric. The triad function of the well-known STREAM benchmark [39] is an essential array operation, also called "linked triad," that computes $C = A + \alpha \cdot B$, where A, B, and C are 1D arrays and α is a ratio between 0 and 1. To facilitate the HBM exploitation and for programming productivity, a single kernel is written using a subset of the available HBM2 channels, followed by replicating the kernel. The kernel is coded inside a macro and calls it multiple times to achieve the replication. Each kernel instance would use a different set of HBM2 channels, with a different kernel name and specifying different buffer_location attributes.

```
1
2 #define NUM_KERNEL_INSTANCES 32
3 #define MY_KERNEL(kernel_name, global_mem_label) \
4 __kernel void __attribute((reqd_work_group_size(1024,1,1))) \
5 kernel_name ( \
6     __global __attribute((buffer_location(global_mem_label))) float8 *
          restrict sA, \
7     __global __attribute((buffer_location(global_mem_label))) float8 *
          restrict sB, \
8     __global __attribute((buffer_location(global_mem_label))) float8 *
          restrict dst, \
9     const float alpha \
10 ) { \
11     uint gid = get_global_id(0); \
12     float8 vA = sA[gid]; \
13     float8 vB = sB[gid]; \
14     vA = vA + alpha*vB; \
15     dst[gid] = vA; \
16     }
17 //support for up to 32 kernels
18 #if NUM_KERNEL_INSTANCES >= 1
19 MY_KERNEL(k0, "HBM0")
20 #endif
21 #if NUM_KERNEL_INSTANCES >= 2
22 MY_KERNEL(k1, "HBM1")
23 #endif
24 #if NUM_KERNEL_INSTANCES >= 3
25 MY_KERNEL(k2, "HBM2")
26 #endif
27 . . .
28 #if NUM_KERNEL_INSTANCES >= 31
29 MY_KERNEL(k30, "HBM30")
30 #endif
31 #if NUM_KERNEL_INSTANCES >= 32
32 MY_KERNEL(k31, "HBM31")
33 #endif
```

Figure 2.4: OpenCL FPGA kernel for triad example.

2.3. Programming models for heterogeneous architectures

To achieve performance productivity in this work, it is worth targeting open, standard-based programming models for heterogeneous architectures rather than proprietary device-specific programming models such as CUDA [40]. This section

presents an overview of the two open programming models used in this work: OpenCL and oneAPI-SYCL.

2.3.1. OpenCL

OpenCL (Open Computing Language) is an open, royalty-free standard for cross-platform, parallel programming of various accelerators found in supercomputers, cloud servers, personal computers, mobile devices and embedded platforms. OpenCL was created by the Khronos Group [41]. The OpenCL API specification enables each chip to have its OpenCL drivers tuned to its specific architecture. Having standardized API available on many systems enables developers to reach more customers while minimizing porting and development costs.

OpenCL is a programming framework and runtime that enables a programmer to develop codes that can be compiled and executed, in parallel, across any processor in a system. The processors can be any mix of different types, including CPUs, GPUs, TPUs, FPGAs, DSPs, as can be seen in Figure 2.5.

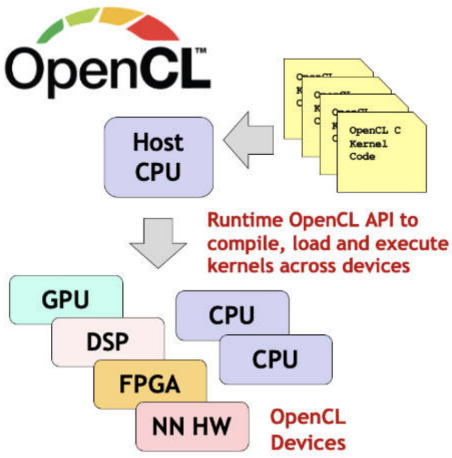


Figure 2.5: OpenCL compile and execute kernels flow overview [41].

An OpenCL application is split into host code and device kernel code. Host code is typically written using a general programming language such as C or C++ and compiled by a conventional compiler for execution on the host CPU. A kernel program is the basic unit of executable code, similar to a C function. Kernels can execute with data or task-parallelism. The most common language for pro-

gramming the kernels is OpenCL C, which is based on C99 and is defined as part of the OpenCL specification. Kernels written in other programming languages work in OpenCL by compiling to an intermediate program representation, such as SPIR-V.

OpenCL is a low-level programming framework. This way, the programmer has direct, explicit control over where and when kernels execute, the memory allocation and how the compute devices and host CPU synchronize their operations to ensure that data and computed results flow correctly. The communication between the host and kernels takes place through the command queue. By enqueueing commands into a command queue, kernels and data transfer functions may execute asynchronously in parallel with the host code. The kernels and functions in a command queue can execute in or out of order. A device may have multiple command queues, but a command queue can only communicate with one device.

2.3.1.1. OpenCL Programming Model

The OpenCL programming model is divided into three main parts: Platform, Execution and Memory Models.

The OpenCL Platform Model describes how OpenCL interconnect the compute resources. A host is connected to one or more OpenCL compute devices. A compute device can be, for example, a GPU. Each compute device is a collection of one or more compute units. Each compute unit is composed of one or more processing elements. Processing elements execute code with SIMD (Single Instruction Multiple Data) or SPMD (Single Program Multiple Data) parallelisms.

The OpenCL execution model is divided into the host program and the device code, made of one or more kernels, executed in the compute device. The host interacts with these kernels through command queues, where each device has its command queue. The host submits the kernel code through the command queue, checking for dependencies and executing it. After the execution, the command queue communicates to the host the termination of the kernel life cycle. As a kernel is submitted for execution, the command queue creates an instance for the kernel with an N-dimensional range. A kernel instance is made of the kernel function, the arguments of the kernel and the parameters that define the range. The N-dimensional range (ND-Range) of a kernel is distributed in three levels, as can be seen in Figure 2.6:

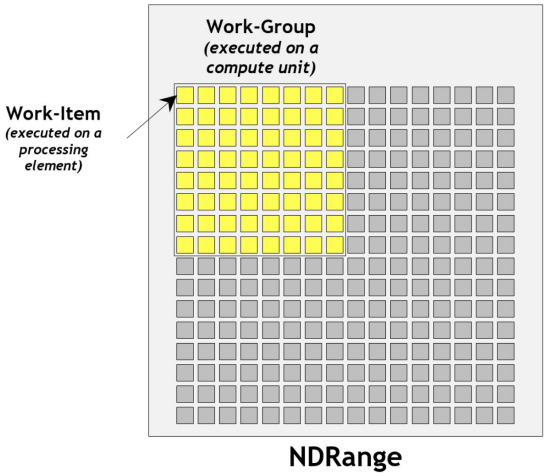


Figure 2.6: ND-Range Compute device Execution Model [42].

- Work-item: is the individual execution of a point of the kernel function. It is also called a thread. It is executed on a processing element
- Work-group: is a 1-,2-, or 3-dimensional set of work-items. It is executed on a compute unit.
- ND-Range: contains the total set of operations to be performed. The ND-Range is distributed among the work-groups witch will be executed in the available compute units.

Individual work-items can be identified by a global ID or a combination of work-group ID and a local ID for that work-group. The work-group runs the same kernel code, capturing the data-parallel computing. A work-group is executed in a device in random order. Also, the work-items within a work-group execute concurrently in random order. Synchronization points between work-items must be precisely defined and controlled by the programmer, as needed. This execution model will be detailed further for different programming models in Section 2.3.

OpenCL has a hierarchical memory model that can be seen in Figure 2.7.

The memory model is breakdown into:

- Host memory: available to the Host (CPU).
- Global/Constant memory: available to all compute units in a device.

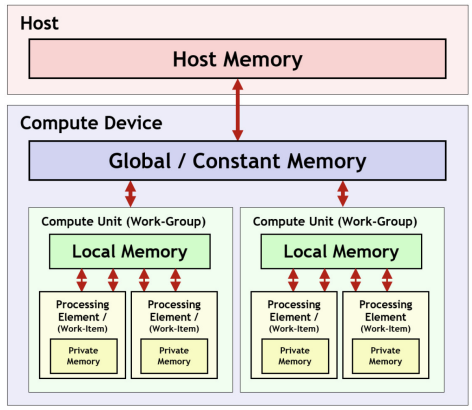


Figure 2.7: OpenCL Memory model [42].

- Local memory: available to all the processing elements in a compute unit.
- Private memory: available to a single processing element.

OpenCL memory management is explicit, which means that the memory do not automatically synchronizes. The application needs to specify the data movement between different memory levels.

2.3.2. oneAPI and SYCL

SYCL is a royalty-free, cross-platform abstraction layer that enables code for heterogeneous processors to be written using standard ISO C++ with both the host and kernel code for an application contained in the same source file [43].

SYCL uses generic programming with templates and generic lambda functions to enable higher-level application software to be cleanly coded with an optimized acceleration of kernel code across the extensive range of various acceleration APIs, such as OpenCL. Developers program at a higher level than the native acceleration API but always have access to lower-level code through seamless integration with the native acceleration API through the interoperability mode, C/C++ libraries, and frameworks such as OpenCV or OpenMP.

SYCL implementations are available from an increasing number of vendors, including adding support for various acceleration API back-ends in addition to OpenCL, as can be seen in Figure 2.8.

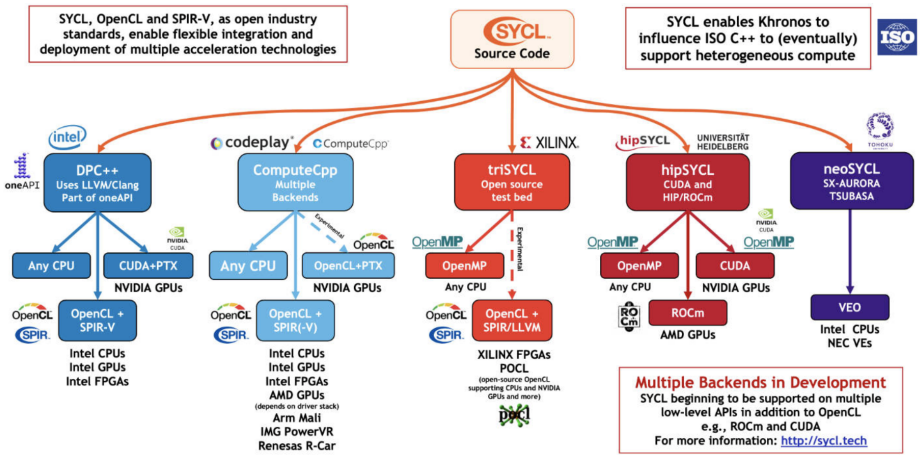


Figure 2.8: SYCL API implementations currently available [44].

We are going to examine the one used in this work, oneAPI [45]. oneAPI is a promising framework that simplifies programming heterogeneous architectures by providing several libraries and an unified programming language DPC++ (Data Parallel C++) [46] able to target CPUs, GPUs and FPGAs. DPC++ is an open-source language based on modern C++ and the SYCL [43] from the Khronos Group with some additional Intel extensions. DPC++ incorporates extensions for Unified Shared Memory (USM), ordered queues, reductions, subgroups (on CPU and GPU implementations), and data flow pipes (for FPGAs) support. The main benefit of using oneAPI over other languages, as OpenCL or CUDA, is the single programming language approach, which enables targeting multiple platforms using the same programming model and, therefore, having a cleaner, portable, and easier to maintain code. The host and offloaded accelerator codes can be combined in a single source file. Likewise, the syntax is standard C++: no new keywords or pragmas are used to express the parallelism. Instead, parallelism is expressed through C++ classes. The accelerator (such as a CPU, GPU or FPGA) to offload the code is selected in the host code using a device queue. The queue is initialized in host code using different flags: (1) *default_selector*, an heuristic chooses among the available devices the most suitable one; (2) *cpu_selector*, the CPU is selected as an accelerator; (3) *gpu_selector*, to target the GPU; (4) *ext::intel::fpga_emulator_selector* to target CPU as emulation of FPGA, (5) *ext::intel::fpga_selector*, to target a real Intel's FPGA; (6) *accelerator_selector*, that can be customize for target other generic devices.

It is worth mentioning that, although DPC++ applications can run on any supported target hardware, tuning is required to derive the best performance advantage on the given target architecture. For example, code tuned for a GPU likely will not run as fast on a CPU selected as an accelerator.

Single source compilation has several benefits compared to separate host and device code compilation. The oneAPI programming model supports single-source compilation. It should be noted that the oneAPI programming model also supports separate host and device code compilation. Advantages of the single source compilation model include:

- Usability, programmers need to create fewer files and can define device code right next to the call site in the host code.
- Extra safety, single source means one compiler can see the boundary code between host and device and the actual parameters generated by host compiler will match formal parameters of the kernel generated by the device compiler.
- Optimization, the device compiler can perform additional optimizations by knowing the context from which a kernel is invoked. For instance, the compiler may propagate some constants or infer pointer aliasing information across the function calls.

In a program with an offload computation, the compiler must generate code for the host and the device. oneAPI tries to hide this complexity from the developer. A Data Parallel C++ (DPC++) application is compiled with *dpcpp* compile command and generates the host and device code. The compilation flow for DPC++ offers two options for the device code: Just-in-Time (JIT) compilation and Ahead-of-Time (AOT) compilation. In the JIT compilation flow, which is the default one, when the application is running the runtime determines the available devices, the target device with the device queue and generates the code specific to that device. This allows for more flexibility than the AOT flow, which must specify a device at compile time. However, performance may be worse because compilation occurs when the application runs. Larger applications with substantial amounts of device code may notice performance impacts due to longer compilation times. FPGAs differ from CPUs and GPUs in generating device code. Generating the device binary for the FPGA hardware is a computationally intensive and time-consuming process. An usual FPGA compilation takes several hours to complete, which makes impractical just-in-time (or online) compilation. For this reason, only ahead-of-time (or offline) kernel compilation mode

is supported for FPGA. The Intel oneAPI DPC++/C++ Compiler provides several mechanisms like emulating FPGA on CPU, partial compilation or simulation, enabling a quick iteration on FPGA designs.

2.3.2.1. oneTBB and FlowGraph

oneAPI API-based programming is supported via sets of optimized libraries included in the oneAPI product, pre-tuned for use with any supported target architecture, eliminating the need for developer intervention. For example, the BLAS routine available from Intel oneAPI Math Kernel Library is just as optimized for a GPU target as a CPU target. API-based programming takes advantage of device offload using library functionality, which can save developers time when writing an application. In general it is easiest to start with API-based programming and use DPC++ offload features where API-based programming is insufficient for your needs. A summary of the API-based library oneAPI provide are: Intel oneAPI Collective Communications Library (oneCCL), Intel oneAPI Data Analytics Library (oneDAL), Intel oneAPI Deep Neural Network Library (oneDNN), Intel oneAPI DPC++ Library (oneDPL), Intel oneAPI Math Kernel Library (oneMKL), Intel oneAPI Video Processing Library (oneVPL), Intel oneAPI Threading Building Blocks (oneTBB).

The oneAPI Threading Building Blocks (oneTBB) library [47] is a solution for writing parallel programs in C++ which has become the most popular and extensive support for parallel programming in C++. oneTBB provides support for parallelism where the C++ standard has not sufficiently evolved, or where new features are not fully supported by all compilers. It also provides higher-level abstractions for parallelism that are beyond the scope of what the C++ language standard will likely ever include. oneTBB contains a number of features, as shown in Figure 2.9.

The three most common layers of parallelism that are expressed in parallel applications are: The message-driven layer, the fork-join layer and the Single Instruction, Multiple Data (SIMD) layer. The message-driven layer captures parallelism that is structured as relatively large computations that communicate to each other through explicit messages. Common patterns in this layer include streaming graphs, data flow graphs, and dependency graphs. one TBB support these patterns through the Flow Graph interfaces. The fork-join layer supports patterns where the serial computation is split in parallel tasks and then continues in serial when the parallel computations are completed. Examples of fork-join patterns include task parallelism, parallel loops, parallel reductions, and pipelines. oneTBB supports them with its Generic Parallel Algorithms. Finally,

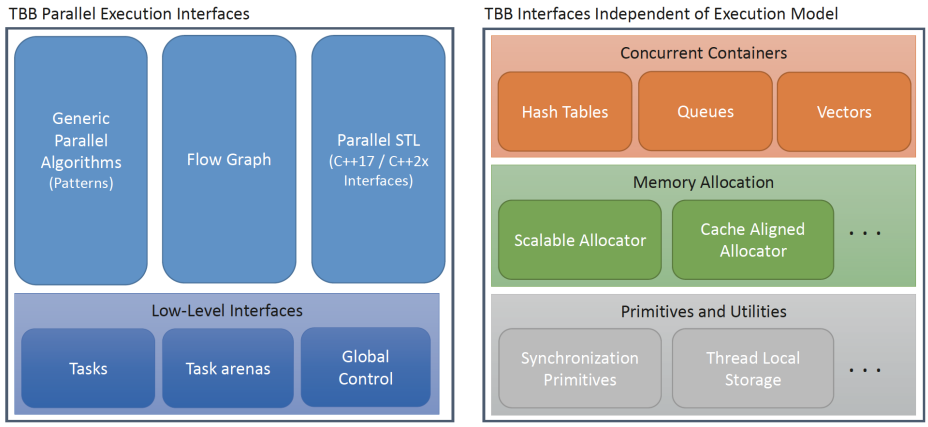


Figure 2.9: oneTBB Library Features [47].

the Single Instruction, Multiple Data (SIMD) layer is where data parallelism is exploited by applying the same operation to multiple data elements simultaneously. This type of parallelism is often implemented using vector extensions such as AVX, AVX2, and AVX-512.

oneTBB potential is truly leveraged when these interfaces are mixed together, a feature called "composability". An example could be an application that uses FlowGraph at the top level with nodes that use nested Generic Parallel Algorithms. These parallel algorithms, in turn, may contain SIMD operations. These parallelisms are exposed to the oneTBB library, which can schedule all the tasks in an efficient way, making the best use of the platform's resources.

The Flow Graph interface The Flow Graph is aimed at applications that react as data becomes available, such as streaming applications, or applications that contain parallelism that can be expressed as graphs. We call these data flow graphs. In many cases, these applications stream data through a set of filters or computations. Graphs can also express before-after relationships between operations, allowing us to express dependency structures that cannot be easily expressed with a parallel loop or pipeline. Thus, the Flow Graph construct can be used to exploit graph parallelism. The Flow Graph interfaces have been successfully used in a wide range of domains including in image processing, artificial intelligence, financial services, healthcare, and games. A Flow Graph is made of the graph object, the nodes and the edges. We first create a graph object. We then create nodes to perform operations on messages that flow through

the graph, such as applying user computations, joining, splitting, buffering, or reordering messages. We use edges to express the message channels or dependencies between these nodes.

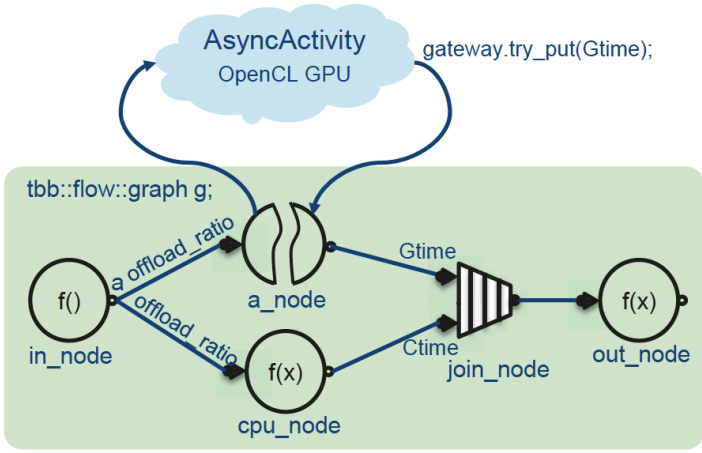


Figure 2.10: Flow Graph Heterogeneous Triad example [47].

Recent features incorporated into the oneTBB library allow for offloading computation to asynchronous devices, embracing heterogeneous computing. This offloading is reached through a new type of node: *async_node*. Figure 2.10 shows an example of Flow Graph executing a Triad [39] benchmark but with a heterogeneous CPU + GPU execution: part of the computation is offloaded to the GPU thanks to the async_node. This node differs from regular function_nodes in providing asynchronous execution. A regular function_node offloading work to an accelerator, such as GPU or FPGA, will cause a blocking function inside a user-level task to block the task and block the OS-managed worker thread processing this task. One factor that makes oneTBB composable is that adding nested levels of parallelism does not increase the number of worker threads, avoiding over-subscription and its associated overheads from ruining our performance. To make the most out of the hardware, oneTBB is configured by default for running as many threads as logical cores. The different oneTBB parallel programs only add enough user-level lightweight tasks to feed these worker threads and exploit the cores. Using the function_node for offloading computation to an accelerator, if we had a worker thread per core and one of them was blocked in the function_-node, the corresponding core may become idle. In such a case, we would not be fully utilizing the hardware.

The async_node solves this issue. When the async_node offloads computation to an accelerator, the worker thread that was taking care of this task switches to work on other pending tasks of the flow graph. This way, the worker thread does not block, leaving an idle core. The Flow Graph is warned about an async_node executing an asynchronous task, keeping a flag to wait for its completion. When the task is finished, the results are re-injected into the graph. The node finishes, and the computation continues as expected.

2.4. Data massive applications and their challenges when ported to heterogeneous applications

As detailed in Section 1.3, this thesis focuses on proposing optimal solutions for heterogeneous architectures for two data massive, real-life and complex problems widely used in big data fields: time series and the skyline. This section covers an introduction, the literature background of these two applications, the state-of-the-art and the challenges for heterogeneous implementations.

2.4.1. Time Series Application: The Matrix Profile

A time series is a collection of sequentially taken observations, as the electrocardiogram example depicted in Figure 2.11.

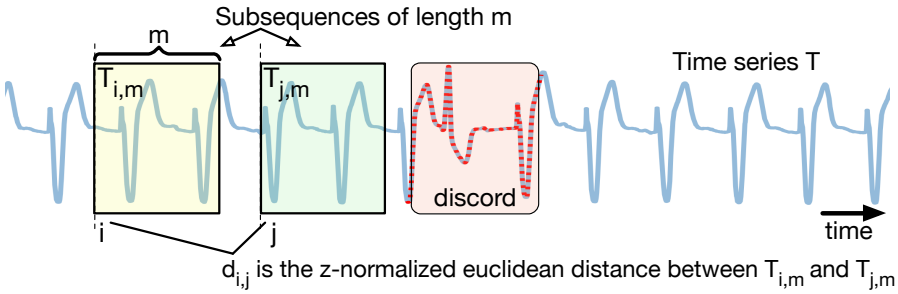


Figure 2.11: Electrocardiogram time series T and two subsequences from which we can compute the distance $d_{i,j}$. The goal is to find motifs/discords, as the ventricular arrhythmia highlighted in the red box. Notation is defined in Section 3.1.

Time series analysis covers many fields, such as cloud computing [48, 49, 50], forecasting [51, 52, 53, 54], clustering [55, 56], similarity search [57], geology [58], geodesy [59], or economics [60, 61].

In particular, the discovery of similarities (motifs) or critical points (discords) in a time series is relevant for several of the previous problems. Motifs and discords can be found via probabilistic approaches [62, 63], machine learning [64] or proposing spatio-temporal models [65]. However in this research we focus on the matrix profile [66] alternative because it provides an exact solution that can not be obtained by probabilistic approaches.

Innovative implementations have been proposed for the matrix profile computation since its first appearance as the *STAMP* [66] algorithm, such as *STOMP* [20], *SCRIMP* [21] and *SCAMP* [67].

STAMP algorithm [66] presents an $O(n^2 log(n))$ complexity (being n the length of the time series) thanks to an FFT based computation of the dot product of two subsequences (which is needed to find the distance/similarity between them [68]). On the positive side, the rows of the matrix profile can be computed in any order and therefore in parallel.

STOMP's [20] complexity is reduced to $O(n^2)$ at the cost of a sequential traversal of rows of the matrix profile. However, the diagonals can be computed in parallel and this is the main property exploited in *SCRIMP*.

SCRIMP also been implemented in different parallel architectures, such as farms of GPUs [22, 67], a distributed-memory implementation aimed at multi-dimensional time-series [69], distributed memory multi-computers [70] and some optimizations proposed for the execution on Intel Xeon Phi KNL processors that integrate 3D-stacked high-bandwidth memory (HBM) [71, 72].

The state-of-the-art algorithm to compute the matrix profile, *SCAMP* [67] takes advantage of the Pearson correlation to compare subsequences, instead of euclidean distance. The use of the Pearson correlation improves both performance in the computation and accuracy in the results.

Practical implementations of matrix profile have been developed. On the one hand, from a practitioner point of view, the research by [73] provides an easy-to-use framework to contextualize the computation of matrix profile along different domains. In [74] matrix profile results are compared and analyzed with other state-of-the-art algorithms to prove their strengths in performance and programming effort. Likewise, a framework for indexing the matrix profile allows arbitrary range queries and avoids recomputing the entire matrix profile in case of needing a particular range of queries [75]. On the other hand, from an accuracy

point of view, all previous algorithms use z-normalized euclidean distance to measure the distance between subsequences. However, this distance can report wrong results for some datasets, as it is pointed in [76], where AAMP and ACAMP algorithms are proposed to improve accuracy using non-normalized and normalized euclidean distance respectively. Motifs and discords are very sensitive to the subsequence length. Previous implementations need to fix this length, which implies previous knowledge of the time series in order to find the right motif and discords. In this regard, [77, 78] propose a framework to compute time series and find motifs and discords in a given range of lengths.

It is not easy to find related work in which Matrix Profile is computed on CPU+GPU or FPGA devices. A CPU+GPU platform was exploited in [79] in order to optimize time series queries. Besides not being related to motif/discord discoveries, their tool leverages functional parallelism (some tasks running on GPU and others on CPU), whereas in our work we tackle the load balancing problem when data parallelism is exploited and both devices work simultaneously on the same data space. Besides, the authors in [80] do not consider scheduling strategies, a key topic in our research.

FPGA architectures are gaining momentum in HPC and Data Centers as an energy-efficient alternative to CPUs and GPUs for different kind of time series problems like neural networks forecasting [81], in chaotic time series predictions[82, 83], in image processing [84], and in cloud servers [85]. However, to date and to the best of our knowledge, there is no heterogeneous implementation for time series problems, like the matrix profile, able to efficiently distribute the computation among CPU cores and accelerators such as GPU or FPGA, which we address in this thesis.

2.4.2. Optimization application: The Skyline operator

The skyline, initially introduced in [86], is an optimization operator widely used for multi-criteria decision making. It allows to minimize a n-dimensional dataset into the smallest subset, usually using as a reduction metric the *minimum* value for each dimension.

Figure 2.12 shows a toy example of a dataset and its corresponding skyline. The skyline is the subset of points that are not eliminated (or not dominated) by any other point in the dataset. Point C has lower values in its two dimensions with respect to point A, thus point A is eliminated from the skyline by point C. Point B has lower value in x than point C, but higher in y, so neither can eliminate the other (they are incomparable), so both B and C end up in the

Point	x	y
A	3	3
B	1	3
C	2	1

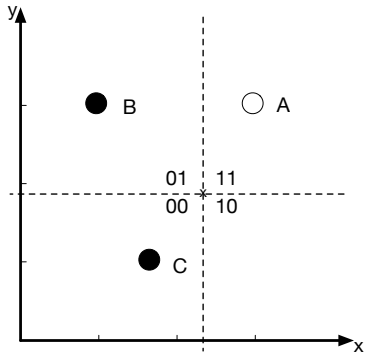


Figure 2.12: Dataset and skyline example. Skyline points are B and C. The numbering 00, 01, 10, 11 corresponds to the nomenclature of a possible space partitioning of the dataset.

skyline set. For large datasets with points of several dimensions, the computation of the skyline becomes a computationally expensive task.

The skyline operator has been applied in the context of privacy-preserving skyline computation framework across multiple domains [87], the processing of skyline query over encrypted data in Cloud-enabled databases [88], an optimization of Quality-of-Services-aware big service processes with discovery of skyline services [89, 90], or a resilient drone service composition framework for delivery in dynamic weather conditions [91].

It also has been applied in the context of reinforcement learning for improving adaptive scheduling of cloud computing services [92], or over multi-source encrypted data for efficient privacy-preserving data merging [93], or evolutionary mining of graph clustering [94].

In addition, generalizations of skyline computation have been proposed in [95], where they work with all the $2^d - 1$ skyline query subdomains combinations from a dataset of d dimensions.

Considering the literature on optimizing a regular skyline computation, initial implementations of the skyline operator [86] were based on the divide-and-conquer approach. Early improvements to the algorithm came in the form of tree-based indexing methods, B-trees [96] and R-trees [97]. In general, the optimizations of this algorithm focus on reducing the number of operations, that

typically are comparisons to asses if a point is in the skyline or not. For it, two main strategies are adopted: sorting and space partitioning.

Sorting-based algorithms [98, 99, 100] introduce a data precomputation phase before entering the main loop. This precomputation phase facilitates both the elimination of points that do not belong to the skyline and the reduction of comparisons between points. Researchers have used different sorting strategies, such as the Manhattan norm [98], z-order [99] or the minimum as a sorting attribute for each dimension [100]. On the other hand, partitioning-based algorithms [101, 102, 103] divide the data space into regions, so that they avoid point comparisons between entire regions, which reduces the number of operations. These strategies typically are based on recursive methods such as pivot point-based partitioning [101], being *BSkyTree* [102] the current state-of-the-art for sequential skyline computation.

The first multi-core parallel approach [103] partitioned the space into blocks. *Hybrid* [23], the state-of-the-art in multicore algorithms, proposes a hybrid strategy: it applies sorting followed by partitioning with no recursion. This algorithm builds dynamically a two-level quad-tree in tiled batches to support multithreading. Algorithms optimized for GPU avoid synchronisations to achieve very high compute throughput. As authors of [24] noticed, *Hybrid* requires dynamical synchronization points to update the skyline, stalling performance of the GPU.

The first algorithm for skyline computation on GPU, *GNL* [104], assigns a point in the dataset to each thread without further optimization. *GGS* [105] improves on this implementation by adding a preprocessing stage that sorts the points according to the Manhattan distance. *SkyAlign* [24], which represents the current state-of-the-art for GPU, outperforms previous works by building a statically-defined quad tree. It also uses medians and quartiles of each dimension to construct virtual pivot points, which are defined globally to create more predictable tree traversals that minimise branch divergence. By contrast, *Hybrid* only uses medians. Authors in [106] propose an alternative to *SkyAlign* for high dimensional datasets and skylines variations with more relaxed rules for pruning points. In any case, in our work we want to focus on a general approach for computing the skyline for arriving data queries with any number of dimensions. In addition, the experimental validation of *SkyAlign* in [24] demonstrates that the parallel scalability of this algorithm is preserved when increasing the number of computing units, what makes it a good candidate for heterogeneous implementations, one of the goals in this work, so we keep *SkyAlign* as our algorithm of reference on the GPU. However, as we will show in Chapter 5, on our platform and for specific datasets, running *Hybrid* on the multicore CPU

outperforms *SkyAlign* on the GPU. Thus, *Hybrid* is still considered for the CPU implementation in some of our heterogeneous proposals.

In any case, another goal of this work has to do with providing support for the skyline computation over a stream of independent data queries. There has been previous research on incrementally computing the skyline for a data query for which data points arrive over time in streams [107, 108, 109, 110, 111, 112]. Typically, the output is a sequence or incremental update of skyline computations. Sequential solutions for continuous data streams maintain a sliding window of the most recent points [107, 108, 109]. Researchers in [110] propose parallel implementations of a previous proposal in [108]. Authors in [111, 112] present parallel solutions on distributed systems for the sliding window approach.

Nevertheless, these approaches focus on considering a stream of dependent datasets of points to process. They need to keep and update a global skyline over time with the historical data processed to be compared with the new queries. Our work, however, considers as an input a stream of independent datasets of points, producing as an output a stream of independent skylines, one per input received.

Parallel implementations of skyline computation over a stream of data queries targeting heterogeneous architectures are still an open research issue. We focus on heterogeneous architectures comprised of a multicore CPU and an integrated GPU. In order to improve performance productivity for this type of architectures, new programming models such as oneAPI [45] or SYCL [43] have been proposed and we consider them for the first time to solve the problem at hand. However, these frameworks do not solve the problem of partitioning the workload and scheduling the tasks among the devices, issues that we address in this work. Thus, to the best of our knowledge, the new heterogeneous proposals introduced here represent novel contributions in this context.

2.5. Strategies for scheduling heterogeneous applications

2.5.1. Heterogeneous scheduling using CPU+GPU

Developing heterogeneous applications that makes the most out of CPU + Accelerator platforms is difficult and error prone due to low-level considerations: data sharing, synchronization, load balancing, scheduling, etc. To make it more

approachable, new programming models and frameworks such as OmpSs [113], HDSS [114], Fluidic [115], oneAPI [116] or SYCL [43] are being proposed.

However, the workload distribution at runtime and specially the selection of the appropriate granularity for each computing device has not received enough attention. In order to address these important issues, our research group has previously proposed a heterogeneous template for the parallel_for pattern [117], which: i) reduces the burden of implementing the heterogeneous application; and ii) is based on an adaptive scheduler that at runtime computes the size of the chunks of work that have to be offloaded to each device to fully utilize them and avoid load unbalance.

Some heterogeneous schedulers have been also proposed in order to distribute the iteration space of a parallel_for among CPU cores and a GPU. The most straightforward one, usually called *Static*, does a single partition, dividing the iteration space in as many chunks as computing devices: usually a big chunk of iterations is offloaded to the GPU and the rest is assigned to the CPU cores. A good enough static partition can be computed by an offline search phase, for example changing the percentage of iterations offloaded to the GPU from 0% to 100% in steps of 10% (which results in 11 runs) and selecting the fastest partition for future runs. A more convenient alternative conducts a single offline run to measure the relative speed of the GPU over the CPU and uses this ratio to do a single partition of the iteration space accordingly. Concord [118] takes a further step by measuring the devices relative speed online instead of offline: some initial chunks of the iteration space are computed on the GPU and CPU and execution times are taken in order to estimate the relative speed and proportionally distribute the remaining iterations. This may work well for regular codes in which the first (profiled) iterations are representative of the rest of the computation. Concord approach is improved in the Maat library [119] that includes a heterogeneous guided (HGuided) scheduler that results in better load balance because the chunks of iterations are larger at the beginning of the iteration space and smaller towards the end.

However, none of these previous approaches consider that larger chunks of iterations offloaded to the GPU can result in less throughput than smaller ones. That observation was considered in the *Dynamic* and *LogFit* heterogeneous schedulers presented in our previous work [117] and validated for irregular codes in which data divergence is an issue and/or the computational load of each parallel iteration is different. These schedulers are especially suitable for architectures with integrated (on-chip) GPU where the overhead of communicating data between CPU and GPU is small or even negligible if Shared Virtual Memory, SVM, is available. Within this framework, offloading several smaller chunks of iterations

to the GPU can be profitable w.r.t. offloading just a single larger chunk. In the *Dynamic* scheduler a user provides the GPU chunk size that is used to continuously feed the GPU, whereas the CPU cores are fed in parallel with CPU chunks. On the contrary, the *LogFit* scheduler does not require user intervention because at runtime computes a near-optimal GPU chunk size with an algorithm inspired in the HDSS scheduler [120], as well as the optimal CPU chunk size that ensures load balance. *LogFit* is designed as a three-phase partition strategy: the Exploration Phase, the Stable Phase and the Final Phase. Summarizing, in the Exploration Phase the throughput for different GPU chunk sizes is examined and a logarithmic fitting is applied to find the GPU chunk size that maximizes the throughput. In the Stable Phase, GPU throughput is monitored and adaptive GPU chunk sizes are re-adjusted following the previously computed logarithmic fitting. Finally, when there are few remaining iterations, the Final Phase pays especial attention to the load balance.

All these heterogeneous scheduling strategies are examined in this work for the data massive applications of interest, and new more efficient scheduling proposals are introduced as we will see in the next sections.

Another parallel pattern that we tackle in this thesis is related to data flow applications in which a set of dependent computations are carried out over a stream of independent data inputs: parallel data flow. An important challenge for this pattern is to exploit graph parallelism on heterogeneous CPU + GPU architectures, an issue that we address in this thesis. In particular we propose novel policies for scheduling the computation of arriving data inputs between devices, introducing a new model that at runtime estimates the computation time of any arriving input on each device to enqueue the task on the device where the system throughput is maximized. We also propose a dynamic workload partition of each arriving data input between the GPU and the CPU, and discuss the streaming scenarios where each approach achieves the best performance.

2.5.2. Heterogeneous scheduling using CPU+FPGA

High-level design environments, such as OpenCL, have increased significantly the productivity in FPGA designs [121]. Heterogeneous developments based on CPU + FPGA are more visible every day in many domains, such as astrophysics [122], or clusters and cloud services [123, 124, 125]. Likewise, they have been applied for algorithms optimization, like hash-join [126], NP-hard problems [127], Synthetic Aperture Radar (SAR) simulation algorithm [128], Agent-based Simulations (ABSs)[129], Sparse Matrix Operations [130], Sparse Linear

Algebra[131], or Temporal Convolutional Networks (TCNs) [132]. The problem of using GPU + FPGA at the same time is tackled by [133, 134]. Multi-FPGA to optimize task graphs using heuristics are studied in [135].

Heterogeneous platforms based on CPU + FPGA can be more effectively used if the scheduling of the workload between devices is considered. There are a few application-independent frameworks, e.g., EngineCL [136]. Other scheduling strategies are proposed for different applications, such as Proximal Policy Optimization(PPO) algorithm for reinforcement learning [137], Feature-Aware Task Scheduling [138], Accelerating Graph Convolutional Networks (GCNs) Training [139], Cloud platforms for big data applications [140], or Convolutional Neural Networks, which apply double-buffer to overlap computation and data transfer [141]. The research by [142] proposes a tool for resource estimation in FPGA kernels within the context of an heterogeneous scheduling scheme. In [143], the scheduling is proposed with a mathematical model focusing on minimizing the communication delays. Other interesting approach for scheduling is proposed in [144], where the utilization of CPU + FPGA is maximized by dynamically scaling the resource allocation (FPGA partial reconfiguration) for tasks transparently.

Innovative contributions with different scheduling approaches are shown in [145], which implements *Logfit* and *Dynamic* schedulers presented in [146]. Additionally, [145] proposes a framework based on interruptions to remove spin-locks in the FPGA. The research by [147] presents a framework with a productive and energy efficiency programming model targeted to low-cost and low-power platforms.

However, the FPGA architecture is better exploited by regular algorithms and, in these cases, an adaptive scheduler introduces unnecessary overhead. In this work, we propose the *Fastfit* scheduler, designed for regular codes and tailored to make the most out of the FPGA features by minimizing the scheduling overheads. Although our target Time Series application is irregular, it can be regularized to take advantage of FPGA capabilities. One important feature of the testbed platform is the FPGA support for High Bandwidth Memory (HBM) that is paramount for memory bound applications, like most of the Time Series algorithms. To the best of our knowledge, there are no published proposals to model and optimize the execution of Time Series applications on HBM-enabled FPGAs, which is another of the original contributions of this thesis.

3 Time series on heterogeneous CPU + GPU processors

The discovery of similar subsequences (motifs) or anomalies (discords) in large time series is a computationally expensive problem with applicability in many fields, such as seismology [148], power demand [149], neuroscience [150] or entomology [151], to name a few. A recently proposed solution [66] consists in first computing a score for each subsequence in the time series, resulting in another time series called matrix profile. By a simple inspection of the matrix profile it is straightforward to identify the motifs and discords by focusing on minimum and maximum values respectively. Several algorithms have been proposed to compute the matrix profile: STAMP [66], STOMP [20], SCRIMP [21] or the more advanced SCAMP [22] which exhibits a higher degree of parallelism. Also, different versions of SCRIMP have been adapted to target shared-memory multiprocessors, distributed-memory computers [70], multi-GPU platforms [22] and Intel Xeon Phi KNL processors [71].

This chapter proposes an heterogeneous CPU + GPU implementation of the Matrix Profile using SCRIMP algorithm [21] as a starting point. We pay particular attention to the load unbalance problem posed by the SCRIMP algorithm, where each parallel iteration has a different computational workload. As we explain later, this is a consequence of the computational pattern which follows a diagonal traverse of a triangular matrix where each diagonal has a different length. To solve this problem, using Threading Building Blocks [152], TBB, we evaluate the efficiency of a scheduler that relies on a dynamic partition of the

iteration space and compare it with a static distribution based on an analytical partitioning of the workload.

Furthermore, we contribute with a heterogeneous version of the Matrix Profile that distributes the matrix profile computation among CPU cores and an OpenCL capable GPU. To that end, we leverage a previously developed parallel_for template [117] that is based on a heterogeneous scheduler implemented on top of TBB. Our scheduler takes care of the load balance problem among devices, either CPU or GPU, and of finding the appropriate granularity for the chunks of work that are processed on each one. An additional challenge of the heterogeneous implementation is that the kernel of the matrix profile computation consists in a parallel_reduce pattern. Thus, in this chapter we propose some strategies to extend our previous parallel_for template to also implement heterogeneous parallel reductions.

By the time this work was developed, SCRIMP was the state-of-the-art algorithm for computing the Matrix Profile and therefore the one used in this chapter. However, our proposals in this chapter can be applied to general matrix profile computation, independently of the implementation applied: SCAMP or SCRIMP. As we will detail in chapter 4, the main difference between SCRIMP and SCAMP is the way in which the distance between time series points is computed. Hence, our proposal can be easily implemented using SCAMP by simply updating the distance equation.

The rest of the chapter is organized as follows. First we introduce in Section 3.1 the background required to understand the problem at hand. In Section 3.2 we describe the parallel and heterogeneous implementations we propose to optimize the Matrix Profile. Next, the experimental results are presented in Section 3.3. Finally, we wrap up with conclusions in Section 3.4.

3.1. Matrix Profile: Theoretical background

A time series is a collection of sequentially taken observations, as the electrocardiogram one depicted in Figure 2.11. More formally and following the notation introduced in [20], a time series T is a sequence of real-valued numbers $t_i : T = t_1, t_2, \ldots, t_n$ where n is the length of T. A subsequence is a local region of a time series. A subsequence $T_{i,m}$ is the region of consecutive values starting at position i and of length m (a window of length m), i.e. $T_{i,m} = t_i, t_{i+1}, \ldots, t_{i+m-1}$.

We are interested in comparing each subsequence $T_{i,m}$ with all the other subsequences $T_{j,m}$ of the same time series T, with $1 \leq i, j, \leq n - m + 1$. To that

end, we compute the distance profile as the vector $D_i = [d_{i,1}, d_{i,2}, \ldots, d_{i,n-m+1}]$, where $d_{i,j}$ is the z-normalized Euclidean distance between $T_{i,m}$ and $T_{j,m}$:

$$d_{i,j} = \sqrt{2m\left(1 - \frac{Q_{i,j} - \mu_i \mu_j}{m\sigma_i \sigma_j}\right)} \tag{3.1}$$

Here μ_i is the mean of $T_{i,m}$, σ_i is the standard deviation of $T_{i,m}$, and $Q_{i,j} = \sum_{k=0}^{m-1} t_{i+k} t_{j+k}$, i.e. the dot product of $T_{i,m}$ and $T_{j,m}$. Note that once $Q_{i-1,j-1}$ is computed, $Q_{i,j} = Q_{i-1,j-1} - t_{i-1}t_{j-1} + t_{i+m-1}t_{j+m-1}$, so $d_{i,j}$ depends on $d_{i-1,j-1}$.

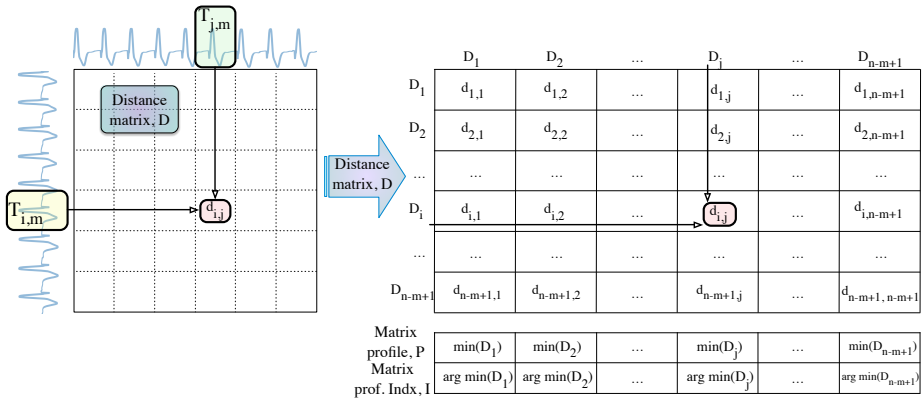


Figure 3.1: Distance matrix, D, matrix profile, P and matrix profile index, I.

The distance matrix, $D = [D_i] \quad (1 \leq i \leq n - m + 1)$, contains all pairwise distances between all subsequences of T and is represented in Figure 3.1. It is straightforward to demonstrate that this matrix D is symmetric ($d_{i,j} = d_{j,i}$), the values in the diagonal are zero ($d_{i,i} = 0$) and values near the diagonal are close to zero ($d_{i,i\pm k} \sim 0$ for small values of k) since neighbor subsequences are quite similar.

A matrix profile, $P = [min(D_1), min(D_2), \ldots, min(D_{n-m+1})]$, is a vector of the distances between each subsequence $T_{i,m}$ and its closest match (nearest neighbor). Intuitively, the matrix profile contains the minimum value of each column of the distance matrix, hence the name "matrix profile". The smallest values in the matrix profile identify the motifs and the highest values the discords. A matrix profile index, I is used to record where these motifs/discords are. Formally, $I = [I_1, I_2, \ldots, I_{n-m+1}]$ where $I_i = j$ if $d_{i,j} = min(D_i)$ that is computed as

$I_i = arg\ min(D_i)$. Note that to avoid trivial matches due to comparing neighbor subsequences, an Exclusion Zone surrounding the diagonal of D is enforced.

3.2. Matrix Profile Optimizations

With all the definitions detailed in Section 3.1, the Algorithm 1 sketches the Matrix Profile computation. From lines 1 to 4 we initialize n, compute μ_i and σ_i and initialize the matrix profile, matrix profile index vectors and the set of diagonals that have to be traversed. Since matrix D is symmetric and the exclusion zone is avoided, only an upper triangular matrix is traversed. The elements $d_{i,j}$ of matrix D are not stored, but computed as d in line 10 and later discarded after they are used to update P and I in lines 11-12. Although $d_{i,j} == d_{j,i}$, $P_i = min(D_i)$ can be different to $P_j = min(D_j)$, and the same for I_i and I_j, so the two checks of lines 11-12 are necessary.

Algorithm 1: The Matrix Profile algorithm

Data: A time series T, m and ExclusionZone
Result: Matrix Profile P and matrix profile index I

```
1  n ← Length(T)
2  μ, σ ← preComputeMeanStd(T, m)
3  P ← inf, I ← zeros
4  Diagonals ← (ExclusionZone + 1 : n − m + 1)
5  for k in Diagonals do
6  |   for i ← 1 to length(k) do
7  |   |   j ← i + k − 1
8  |   |   if i == 1 then  q ← DotProduct(T_{1,m}, T_{k,m})
9  |   |   else  q ← q − t_{i−1} t_{j−1} + t_{i+m−1} t_{j+m−1}
10 |   |   d ← CalculateDistance(q, μ_i, μj, σi, σj)
11 |   |   if d < P_i then  P_i ← d, I_i ← j
12 |   |   if d < P_j then  P_j ← d, I_j ← i
13 return P, I
```

As we said, the dot product $Q_{i,j}$ can be computed from $Q_{i-1,j-1}$ so the outer loop (line 5) traverses the set of *Diagonals* and the inner one (line 6) each diagonal. For the same reason, the dot product is computed only for the first row (see line 8) and just updated in $O(1)$ for the remaining elements of the diagonal (line 9).

The parallel implementation traverses the diagonals in parallel: several threads can process different diagonals at the same time. The only dependence that needs attention is the update of P and I vectors since two threads traversing two different diagonals can try to simultaneously write in the same P_i and I_i position. A possible solution consists in declaring P and I as vectors of atomics, but a more efficient alternative is based on privatizing both vectors. That way, each thread has a private copy of P and I which avoids conflicting accesses to the vectors and incurs in an acceptable increment of the memory footprint on the platforms that we consider in this paper, in which the number of cores/threads is not large. A reduction step is needed to compute the global P and I vectors, but in our experiments the reduction time represents less than 0.01% of the total execution time.

3.2.1. Multi-core partitioning strategies

As we have seen in the Matrix Profile implementation (Algorithm 1), the outer loop that traverses the diagonals (line 5) can be executed in parallel. However, each parallel iteration exhibits a different workload due to the inner loop (line 6) depends on the diagonal length. In this section, we elaborate on two alternatives to avoid load unbalance: i) an ideal static one that computes the number of diagonals that each device has to process based on an analytical partition; and ii) a dynamic alternative based on the work stealing scheduler provided by the TBB library.

3.2.1.1. Static partitioner

Assuming we have nth threads collaborating on the computation of the matrix profile P, the problem we want to solve is to identify the first and the last diagonal (i.e. the chunk of diagonals) each thread has to process, so that all threads have approximately the same load (number of elements of D). In Figure 3.2, assuming $nth = 4$ and discarding the Exclusion Zone, 4 chunks of diagonals should have about the same number of elements ($A_i == A_j, 0 \leq i, j < nth$).

The threads IDs, tid, are in the range $[0, nth)$ and thread tid_i is responsible of computing the chunk of diagonals $[x_i, x_{i+1})$. The index of the last diagonal is $mpl = n - m + 1$. Figure 3.2 shows that $x_i = mpl - y_i + 1$ and therefore, the problem of finding x_i translates into finding y_i.

The number of elements in the upper-right triangle, A_3 in Figure 3.2, is equal to $\sum_{i=1}^{y_3} i = y_3(y_3 + 1)/2$. Likewise, we say that $N = y_0(y_0 + 1)/2$ is the total

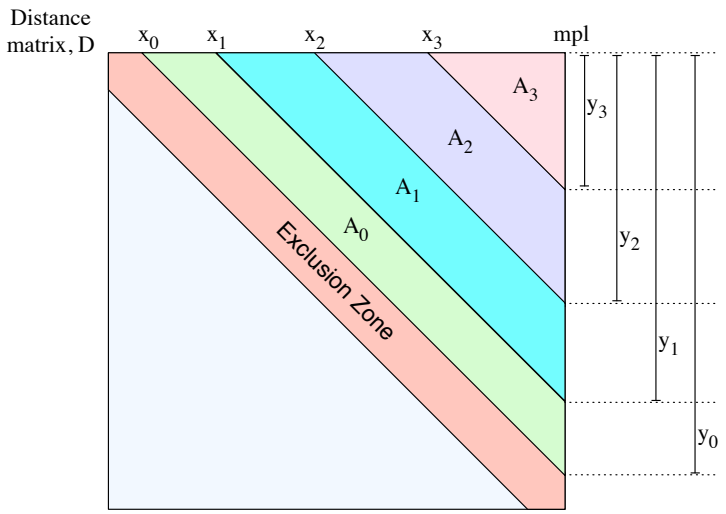


Figure 3.2: Static distribution problem: compute x_i so that the number of elements in each chunk of diagonals, A_i, are approximately the same.

number of elements in all diagonals. Therefore we want that $A_3 = y_3(y_3+1)/2 = N/nth$ (one forth of the number of elements if $nth = 4$). In the same manner $A_2 + A_3 = y_2(y_2+1)/2 = 2N/nth$, $A_1 + A_2 + A_3 = y_1(y_1+1)/2 = 3N/nth$, and by induction $y_i(y_i+1)/2 = (nth - i)N/nth$. Therefore we can solve y_i as 3.2, from which x_i can be computed as we mentioned in the previous paragraph

$$y_i = round\left(\frac{-1+\sqrt{1+8(nth-i)N/nth}}{2}\right) \tag{3.2}$$

With this ideal static partitioning each thread has approximately the same number of elements (± 1 diagonal). In our experiments this translates into a negligible unbalance (less than $10^{-5}\%$).

3.2.2. Work-stealing partitioner

The fact that an ideal static partition can almost perfectly balance the number of elements among all threads does not necessarily imply that all threads/cores will finish at the same time. An alternative work distribution strategy we evaluate is based on the work-stealing policy provided by TBB. On commodity chips

governed by commodity operating systems there are several sources of noise and architectural asymmetries that may result in some cores being more loaded or working at a different speed. In those cases, a dynamic partitioning pays off, although it may add some partitioning overhead. A typical solution consists in a work sharing paradigm in which threads pick up chunks of work from a global queue of tasks. However, this single global queue may represent a contended resource and become a bottleneck. The work stealing alternative [153] tackles this issue by allocating a work queue to each working thread and incorporating a work stealing policy in which threads with an empty work queue can steal work from busier threads.

The TBB library [152] is mainly based on a work stealing scheduler and task-level parallelism. In addition, TBB offers some helpful templates and high level C++ classes to ease the development of parallel applications, so we have implemented the Matrix Profile algorithm drawing on two TBB templates: *parallel_for* and *combinable*.

The `parallel_for` template can dynamically partition the iteration space into chunks of variable size. A given number of working threads is created and fed with chunks (tasks in the TBB jargon). The iteration space is recursively split into chunks and thanks to the work stealing policy each thread's work queue gets populated with enough chunks to keep the thread busy. At the end of the iteration space, some stealing operations may take place to balance the workload. Furthermore, with the default TBB partitioner (*auto_partitioner*) the chunk size can be larger at the beginning of the iteration space and shrink to just 1 iteration when there are few remaining iterations, resulting in a maximum unbalance bound of 1 diagonal. Thus, faster cores (or less loaded) will process more elements than slower ones.

The `combinable` class provides thread local storage (privatization) and includes a `combine` member function that reduces all private copies in order to compute the final result. In our implementation, we hold the matrix profile information in a vector of structs with two fields: a float to store the matrix profile value, P_i, and an integer to store the matrix profile index, I_i, i.e. `struct mp {float val; int idx}`). With this data structure, we can create a `combinable<vector<mp» mp_cpu` object. The object constructor takes care of allocating and initializing the vector. Inside the body of the `parallel_for` each task invokes `mp_cpu.local()` which returns a reference to a thread local `vector<mp>` so it is safe to update the privatized matrix profile values and indices. After the `parallel_for` is computed, a call to `mp_cpu.combine()` will return the reduced matrix profile vector.

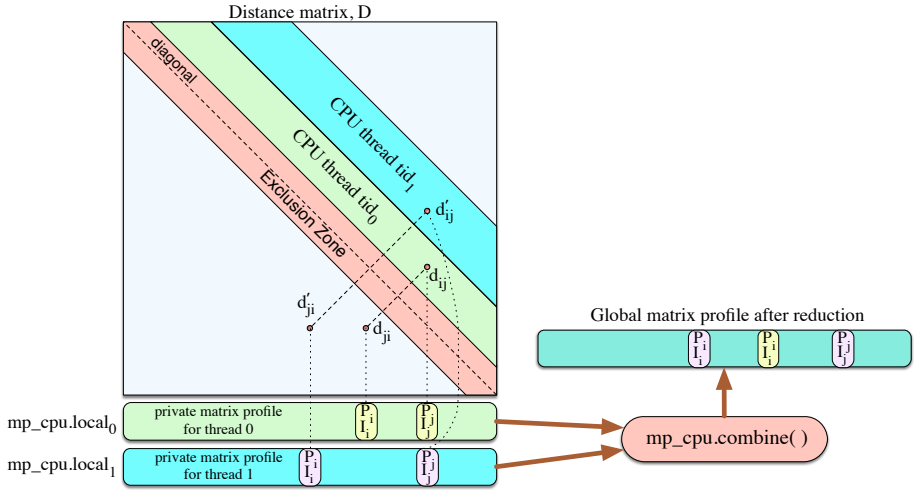


Figure 3.3: TBB implementation of Matrix Profile: `parallel_for` + `combinable`, assuming 2 threads.

As illustrated in Figure 3.3 two threads, tid_0 and tid_1, are collaborating in computing the matrix profile, but each one updates a private copy of `mp_cpu`. This prevent a read-modify-write conflict if tid_0 and tid_1 try to simultaneously update P_j and I_j. The TBB `combine()` member function comes in handy to easily implement the reduction of the private copies.

3.2.3. HetMP: Heterogeneous implementation

Many commodity processors for desktops, laptops and smart phones feature an integrated GPU along with the CPU cores. A heterogeneous implementation that efficiently exploits both devices poses some challenges: (i) Programming language: the CPU and the GPU can have different programming models. In our approach we use OpenCL for the GPU and C++ for the CPU and orchestrating code; (ii) Data sharing: the CPU and the GPU have to share data in order to collaborate. In our approach we leverage the Shared Virtual Memory (SVM) available in OpenCL 2.1 to reduce the communication overheads; and (iii) Workload balance: the CPU and the GPU have to evenly share the computational load. In our approach we adapt and extend a previously developed heterogeneous scheduler [117]. In this section, we elaborate on how these problems have been

addressed and on some details of the heterogeneous implementation of Matrix Profile that we call HetMP.

3.2.3.1. User interface

Figure 3.4 shows an excerpt of the `main` function for the HetMP implementation. Our Heterogeneous Building Blocks, HBB, library encapsulates most of the internal details of the implementation and it is made available by including the `hbb.h` header file (line 1).

```
1  #include "hbb.h"
2
3  int main(int argc, char* argv[]){
4    Params p;
5    InitParams (argc, argv, &p);
6    ...
7    struct mp {float val; int idx};
8    combinable<vector<mp>> mp_cpu;
9    init_CPU_MP(mp_cpu);
10   init_GPU_MP(mp_gpu);
11   preComputeMeanStd(T,m);
12   ...
13   // Instantiate task scheduler
14   auto *hs = {Static, Dynamic, LogFit}::getInstance(&p);
15   Body body;
16   hs->parallel_for(ExlusionZone+1, n-m+1, body);
17   reduce(mp_cpu, mp_gpu);
18   ...
19 }
```

Figure 3.4: HetMP `main()` function.

Program arguments can be read from the command-line, as can be seen in line 5. HetMP accepts four command-line arguments: `<num_cpu_cores>`, `<num_gpus>`, `<timeseries_file>` and `<sch_arg>`. The last argument meaning depends on the particular implementation of the selected heterogeneous scheduler. For instance, the *Static* scheduler requires the `ratioGPU` argument that states the percentage of the iteration space that is offloaded to the GPU, and the *Dynamic* scheduler needs the `gpuChunksize` argument that is the size of the GPU chunks. Initialization chores are carried out in lines 7 to 11, where the `combinable` object, `mp_cpu`, and private GPU buffer, `mp_gpu`, are constructed and initialized and global arrays to store μ and σ are computed. The GPU buffer is implemented as a Shared Virtual Memory, SVM, region. This

is an OpenCL 2.1 feature in which the buffer seats in a region of memory that both the CPU and GPU can read and write. We configure the SVM buffers in the *fine-grained* mode (adding the flag `CL_MEM_SVM_FINE_GRAIN_BUFFER` to the OpenCL call `clSVMAlloc`) so that CPU and GPU cache coherency is maintained by hardware.

Currently, our `HBB` library provides three heterogeneous schedulers, each one with a different partitioning strategy: *Static*, *Dynamic* and *LogFit*. They can be instantiated as shown in Figure 3.4 at line 14, were briefly described in Section 2.5.1, and will be evaluated in Section 3.3.

The `parallel_for()` function template receives three parameters (line 16): the first two parameters describe the parallel iteration space. The iteration space goes from `ExclusionZone+1` to `n-m+1`. The `parallel_for()` scheduler takes care of partitioning this iteration space into chunks of iterations that are assigned to CPU cores or offloaded to the GPU. The third parameter is the `Body` instance that should include the user implementation of the CPU and GPU loop body (this is, two functions that define how a chunk of iterations should be processed on the CPU and on the GPU, respectively). The implementation of the `Body` class is presented in Figure 3.5.

```
1  class Body{
2
3  public:
4    void operatorCPU(int begin, int end) {
5       vector<mp> &priv_mp = mp_cpu.local(); // access a private matrix
            profile
6       for(k=begin; k!=end; i++){
7          ...    // Process diagonal k (lines 6-13 of Alg. 1) updating
               priv_mp
8       }
9    }
10
11   void operatorGPU()(int begin, int end){
12      clSetKernelArg(...,&begin);              // Kernel argument
13      clSetKernelArg(...,&end);                // Kernel argument
14      clEnqueueNDRangeKernel(...,kernel,...); // Launch OpenCL kernel
15      clFinish(...);                           // Wait for kernel
            completion
16   }
17 };
```

Figure 3.5: Body class for HetMP.

The `parallel_for` finishes when all the diagonals have been processed but at that point the results are scattered in several private buffers, one per core (managed with the combinable object, `mp_cpu`), and an additional one for the GPU in a SVM buffer, `mp_gpu`. In line 17 of Figure 3.4 the function `reduce(mp_cpu, mp_gpu)` is invoked in order to combine the partial results and generate the final matrix profile data.

As we see in Figure 3.5, two member functions have to be defined in the `Body` class: `operatorCPU` and `operatorGPU`. The former processes the chunk of iterations `[begin,end)` on a single CPU core. Several threads (one per available CPU core) will be running this function in parallel. Thus, each thread first gets a reference to a private matrix profile vector (line 5). This private vector is safely updated following lines 6 to 12 of Algorithm 1.

The `operatorGPU` member function also processes a chunk of diagonals, but in this case it offloads the work to a GPU. To that end, some OpenCL functions are used: `clSetKernelArg()` sets the kernel arguments (the chunk bounds among them), `clEnqueueNDRangeKernel` launches the kernel to the GPU and `clFinish` waits until the offloaded work is done. In the next section, we delve into the OpenCL kernel implementation for the HetMP algorithm.

3.2.3.2. GPU kernel implementation

The implementation of the OpenCL kernel is based on a CUDA version that was described by Zhu et al. [22] for the SCAMP algorithm. Although our approach is similar, we have adapted the implementation to OpenCL 2.1 to take advantage of the SVM features that do away with the *host-to-device* and *device-to-host* operations that are required in CUDA.

The on-chip GPUs that we used for the experimental evaluation (see next section) include 24 compute units and each one can have 16×7 GPU threads (work items) in flight [154]. Therefore, there can be $24 \times 16 \times 7 = 2,688$ kernel instances being executed concurrently on the GPU. Each GPU thread will process a diagonal of the distance matrix, D, so it is clearly prohibitive to allocate a private matrix profile vector for each GPU thread. The immediate alternative is to rely on OpenCL atomics.

To that end, we first add the flag `CL_MEM_SVM_ATOMICS` to the `clSVMAlloc` call used to allocate the `mp_gpu` buffer. However, it is necessary to guarantee the atomic update of both the matrix profile value, P_i, and the matrix profile index, I_i, but there is not atomic operation able to do that in OpenCL. The workaround consists in packing the float value (32 bits) and the integer index (32 bits) in a

union data type (64 bits) that we call mp_entry in line 7 of the code snippet listed in Figure 3.6.

```
1  #pragma OPENCL EXTENSION cl_khr_int64_base_atomics : enable
2
3  typedef union{
4    float vals[2];
5    int  idxs[2];
6    unsigned long val_idx;
7  } mp_entry;
8
9  inline void AtomicMin(volatile __global unsigned long *mpe, float val,
       int idx) {
10
11   mp_entry old_value, new_value;
12   new_value.vals[0] = val;
13   new_value.idxs[1] = idx;
14   old_value.val_idx = *mpe;
15   while (old_value.floats[0] > val){
16     old_value.val_idx = atom_cmpxchg(mpe, old_value.val_idx, new_value.
         val_idx);
17   }
18 }
19
20 kernel void HetMP(..., global mp_entry* mp_gpu, int begin, int end, ...)
21   {
22     //Get a diagonal from the range [begin,end).
23     int k = get_global_id(0) + begin;
24     //Compute distance d following lines 7 to 10 of Algorithm 1
25     ...
26     //Update GPU matrix profile and matrix profile index if necessary
27     if (d < mp_gpu[i].floats[0]) AtomicMin(&mp_gpu[i], d, j);
28     if (d < mp_gpu[j].floats[0]) AtomicMin(&mp_gpu[j], d, i);
29   }
```

Figure 3.6: OpenCL kernel for HetMP.

The whole mp_entry requires 64 bits that can be seen as two floats, or two integers or a single unsigned long. We declare mp_gpu as an array of mp_entry's. Thus, we can use mp_gpu[i].vals[0] to access P_i and mp_gpu[i].idxs[1] to access I_i. However, the atomic update of both values is done at once using their compound reference: mp_gpu[i].val_idx. To enable 64 bits atomics, the pragma of line 1 is compulsory.

From lines 9 to 18, we detail the AtomicMin function that takes care of safely updating the matrix profile information. It is based on the OpenCL atom_-

`cmpxchg` (atomic compare and exchange) [155] that works as a regular *compare_and_swap*, CAS, operation.

The actual GPU kernel is sketched from line 20 to line 29. In line 23 a single diagonal `k` is assigned to the thread using `get_global_id`. This diagonal is computed on the GPU following lines 7 to 10 of Algorithm 1. However, the update of the matrix profile array, P_i and I_i, is carried out in lines 27 and line 28 using the aforementioned `AtomicMin` function.

In order to measure the impact of the atomic updates, we have also developed an OpenCL kernel for HetMP that does not protect concurrent read-modify-write accesses. Clearly, this approximate version may result in a non exact computation of the matrix profile. In the next section, we discuss how these two versions, called Atomic and NonAtomic, compare.

3.3. Experimental results

3.3.1. Experimental setup

We run our experiments in two platforms that are representative of commodity heterogeneous processors with an integrated GPU.

The processor architecture and software details of each platform can be found in Tables 3.1 and 3.2, respectively.

The performance and energy results reported in this section represent the median value in 5 runs. We should note that the results reported here have been obtained without exploiting the Hyperthreading feature of the Intel processors. For our application we found that spawning two threads per physical core do not render significant benefits. As we explain at the end of Section 3.3.2, this algorithm is heavily memory bound, so having more threads putting additional pressure on the memory (along with the GPU) leads to a slight performance degradation. For instance, enabling Hyperthreading on Coffeelake (16 threads) may degrade throughput up to 5.1% when compared to executions with 8 threads.

Energy readouts were obtained thanks to the Processor Counter Monitor, PCM, library, which reports the breakdown of energy consumed on the CPU, on the GPU and and on the uncore components of the chip. The input datasets are random walk time series, which is a widely known and classical type of time series [156]. We use three time series, with the following sizes: 2^{17} (131072), 2^{18} (262144) and 2^{19} (524188) elements each one, to analyze the impact on the

Table 3.1: Platform details (Coffeelake & Kabylake).

Microarchitecture	**Coffeelake**	**Kabylake**
Processor	i9-9900K 3.6 GHz	i7-7700K 4.2 GHz
Number of cores	8	4
Clock Speed	3.6 GHz	4.2 GHz
Max Turbo Frequency	5.0 GHz	4.5GHz
Main memory	32GB DDR4	32GB DDR4
Cache L3	16 MB	8 MB
Litography	14 nm	14 nm
Max TDP	95 W	91 W
Intel Graphics GPU	UHD Graphics 630	HD Graphics 630
Number of GPU Compute Units	24	24
GPU Base Frequency	350 MHz	350 MHz
GPU Max Dynamic Frequency	1.2 GHz	1.15GHz

Table 3.2: Software details (Coffeelake & Kabylake).

Operating System	Ubuntu 18.04.2 LTS, kernel v5.0
Intel Graphics Compute Runtime for OpenCL	version 19.08.12439
Intel OpenCL	2.1
Intel TBB and VTune	2018 Update 6
Processor Counter Monitor	201902
Intel C++ Compiler	19.0.3.206

processor and memory systems. The window size, m, is 1024 and the Exclusion Zone is $m/4 = 256$ as recommended in the literature [21, 22].

3.3.2. Parallel implementations: Optimizing load balance in the multi-core

We start with the parallels implementations, evaluating the two partitioning strategies presented in Section 3.2.1, which are designed to tackle the load unbalance problem in the context of a multi-core. Thus, we compare an ideal static distribution and a dynamic distribution based on work stealing. Figure 3.7 shows the throughput (elements/ms) when the number of cores increases for

the two partitioning strategies and different input sizes, in Coffelake (above) and Kabylake (below).

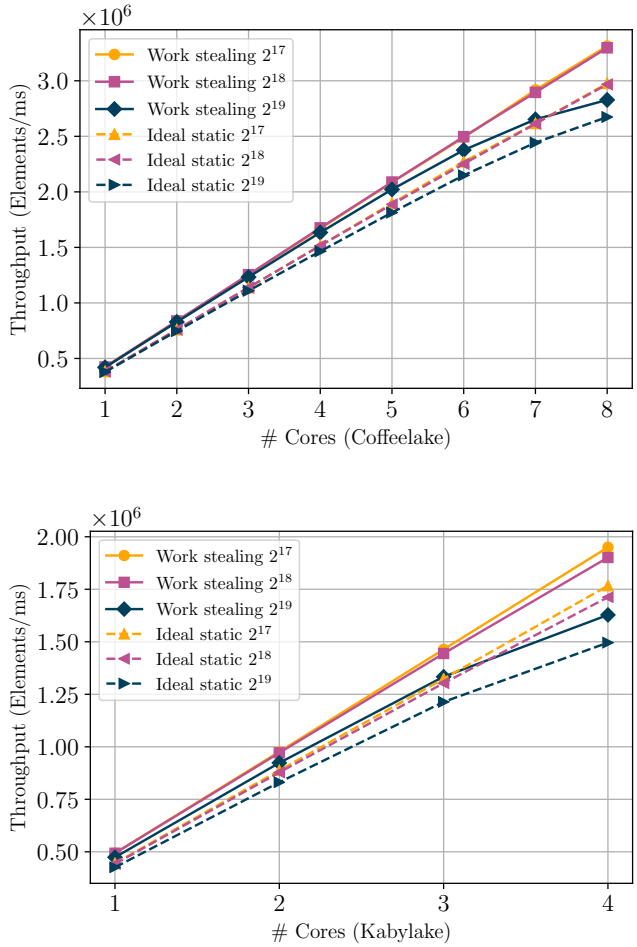


Figure 3.7: Performance scalability of partitioning strategies in the multi-core: ideal static vs. dynamic based on work stealing.

From Figure 3.7, we find that dynamic scheduling based on work stealing always improves the static partitioning performance, in particular when the number of cores increases. For instance, work stealing improves ideal static throughput up to 11.3%, 11% and 5.7% for time series of size 2^{17}, 2^{18} and 2^{19}, respectively,

on the 8 cores of Coffeelake, and up to 10.4%, 11.3% and 8.8% on the 4 cores of Kabylake. Although the static partitioning analytically computes an almost perfect distribution of the diagonals, it can not cope with runtime asymmetries in the load on the CPU cores, even when the experiments were conducted in unburdened scenarios (no other user applications were running on the systems).

When the size of the input grows we also notice that the scalability for both partitioning strategies degrades, although most significantly for the static solution. We elaborate on this issue in more depth, by using Intel VTune, and found that SCRIMP exhibits the characteristics of a heavily memory bound application when the size increases. For instance, in Coffeelake and 8 cores, the percentage of pipeline stalls due to memory demands goes from 0.7% for 2^{17} to 12.8% for 2^{19} in the work stealing experiment, what explains the performance drop (similar measures for Kabylake and the ideal static partitioning).

The parallel implementations were developed during a research stay in the EPCC in Edinburgh. The findings of this study have been reinforced with the results on the ARCHER supercomputer. ARCHER compute nodes contain two 2.7 GHz, 12-core E5-2697 v2 (Ivy Bridge) series processors with 30 MB of Cache L3. Each of the cores in these processors can support 2 hardware threads (Hyperthreads). Within the node, the two processors are connected by two QuickPath Interconnect (QPI) links. Standard compute nodes on ARCHER have 64 GB of memory shared between the two processors. There are a smaller number of high-memory nodes with 128 GB of memory shared between the two processors. The memory is arranged in a non-uniform access (NUMA) form: each 12-core processor belongs to a single NUMA region with local memory of 32 GB (or 64 GB for high-memory nodes). Access to the local memory by cores within a NUMA region has a lower latency than accessing memory on a different NUMA region.

The same experiment performed on Coffeelake and Kabylake has been replicated on an ARCHER compute node. Figure 3.8 shows the throughput (elements/ms) when the number of cores increases for the two partitioning strategies and different input sizes.

In this case, the overall throughput is lower than in Coffeelake and Kabylake due to ARCHER's Ivy Bridge is a weaker processor. For instance, Coffeelake improves the ARCHER node throughput up to 161.47%, 186.62% and 217.56% for 2^{17}, 2^{18} and 2^{19}, respectively, on the 8 cores in work stealing, and up to 142.20%, 157.92% and 201.59% on the 8 cores in ideal static. Comparing the work stealing and ideal static approaches on the ARCHER node, we can see that work stealing improves ideal static throughput up to 3.09%, 3.51% and 3.82% for

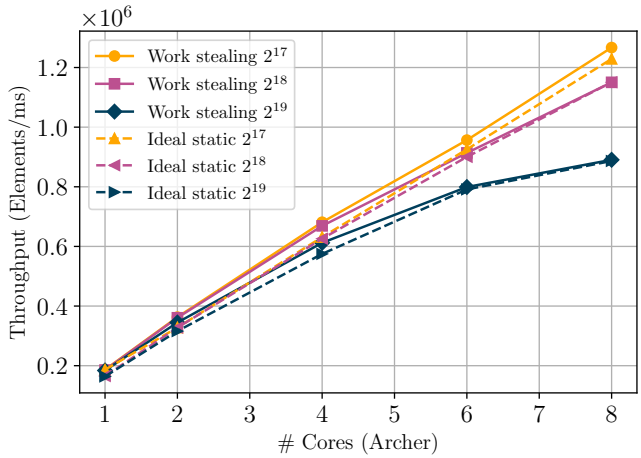


Figure 3.8: Performance scalability of partitioning strategies in the ARCHER node: ideal static vs. dynamic based on work stealing.

2^{17}, 2^{18} and 2^{19}. The relative performance between strategies slightly decreases respect to Coffeelake and Kabylake. That can be explained because the static strategy exploits the cache better due to i) this strategy processes larger blocks of diagonals and ii) Ivy Bridge processors feature a larger L3 cache (30MB vs 16M in Coffeelake and 8MB in Kabylake). However, work stealing is still a better approach than static also for the ARCHER platform.

3.3.3. HetMP: Evaluation of heterogeneous implementations

As explained in Section 3.2.3, we have designed two alternatives for our heterogeneous HetMP algorithm: (i) the **NonAtomic version**, where the matrix profile results, computed on the GPU, are not protected while writing them in the output vector. In this version, data races may affect the accuracy of the final result, although the overhead of atomic contention is avoided; (ii) the **Atomic version**, where the matrix profile results, computed on GPU, are protected while writing them in the output vector. This version guarantees exact results but adds some overhead due to the atomic accesses.

3.3.3.1. Accuracy analysis of heterogeneous implementations

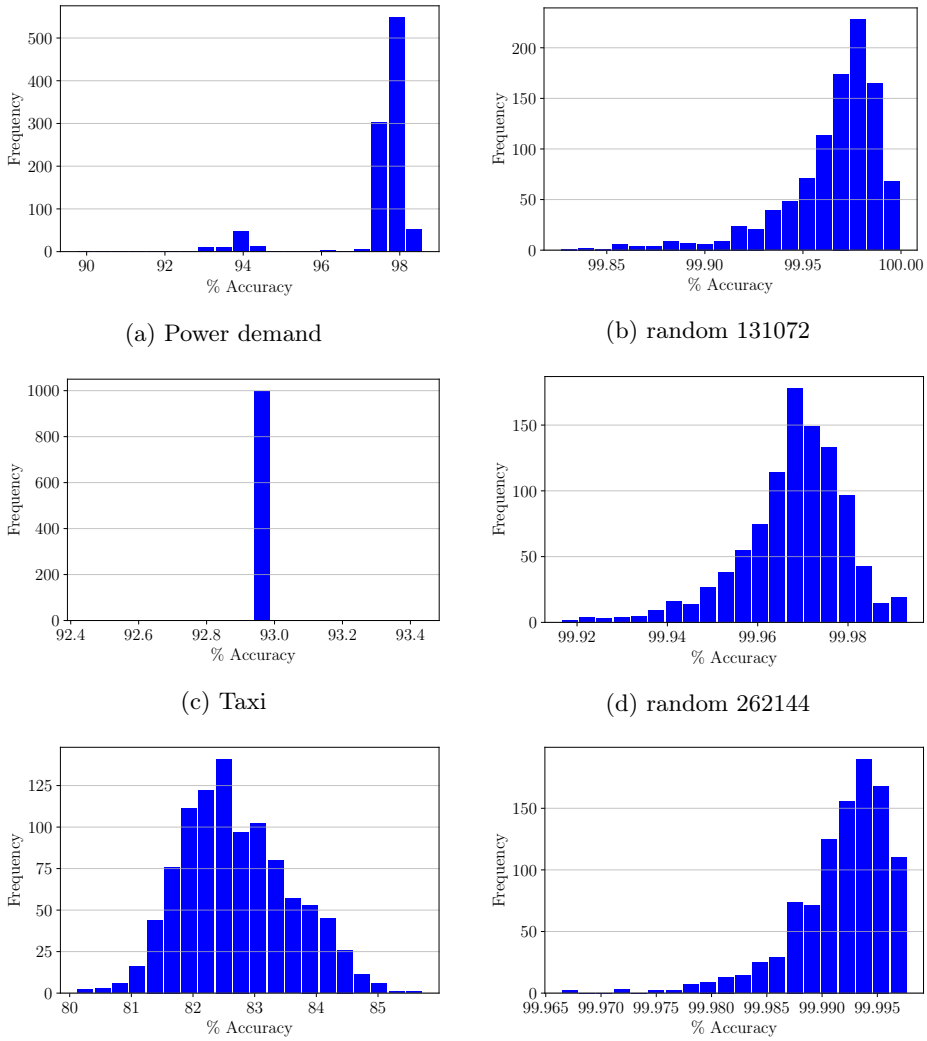
The main goal of having both the Atomic and NonAtomic versions of the HetMP GPU kernel is to assess the overhead introduced by protecting the matrix profile updates with atomics. On Coffeelake and Kabylake, for the GPU only executions, the Atomic kernel reduces the throughput in 8.1% and 8.9% respectively.

For the interested reader, we also study the loss of accuracy of NonAtomic when working with 6 different time series. The three random walk time series presented in 3.3.1 and three real time series of three different scenarios from [157]: Taxi (taxi usage in New York city during a year) with size 3600, PowerDemand (power demand in a house during a year) with size 29931 and ECG (an electrocardiogram) with size 450000. For each time series, we computed the matrix profile 1,000 times with the NonAtomic version and compared the result with the Atomic golden version (that coincides with the one computed sequentially).

Figure 3.9 shows the histogram of accuracy for the 6 time series analyzed.For the three random walk series the percentage of correct entries in the matrix profile was on average 99.96%, 99.96% and 99.99% with a standard deviation of 0.026, 0.011, and 0.004 respectively. For the three real time series the corresponding values were 92.93%, 97.42% and 82.70% with a standard deviation of 0.508, 1.207 and 0.885, respectively. Note, that these last three time series are shorter which results in a smaller matrix profile and a higher probability of collisions of read-modify-write operations, thus the larger impact of the NonAtomic implementation on the matrix profile accuracy. We get the same accuracy on both platforms.

We have also studied the loss of accuracy of NonAtomic when working with our biggest data input: a time series of 2^{19} elements. By comparing the outputs of the NonAtomic and Atomic versions in our runs, we find that matrix profile Index I is always the same in both versions, i.e., the relative position of all motifs and discords is always correct. However, matrix profile P (see Figure 3.1) differs in about 2% of the elements, i.e. in these cases the distance value is not correctly computed in the NonAtomic version. Interestingly, if we check the difference between the unmatched values, we realize that they only differ in an ϵ value less than 10^{-8}, which can be considered irrelevant in many use cases. Actually in our runs the top ten motifs and discords were always the same for the Atomic and NonAtomic versions.

Thus, depending on the use case in the application domain, it could be up to the user selecting the NonAtomic or the Atomic version. As we will see in the



(a) Power demand

(b) random 131072

(c) Taxi

(d) random 262144

(e) Electrocardiogram

(f) random 524288

Figure 3.9: Histogram of accuracy percentage for the 6 time series analyzed. Y-label shows the frequency for each accuracy percentage while X-label shows the distribution of the accuracy percentage.

next section, NonAtomic version provides better performance -but at the cost of this accuracy loss-, while Atomic provides the exact solution.

In the next section, we explore the performance behavior of both versions. Orthogonally, we analyze the behavior of 3 different heterogeneous schedulers that are available in our heterogeneous HBB library (*Static*, *Dynamic* and *LogFit*), which we evaluate in the context of the heterogeneous parallel_reduce solution presented in this chapter. Thus, a total of 6 heterogeneous configurations for our HetMP application are evaluated next.

3.3.3.2. Exploring the behavior of CPU+GPU implementations

Figure 3.10 shows the throughput, in terms of processed elements per millisecond, that NonAtomic (solid lines) vs Atomic (dashed lines) implementations achieve for the *Static* scheduler when the ratio of the iterations offloaded to the GPU, ratioGPU, goes from 0 (0% -only CPU cores are working-) to 1 (100% -only GPU is working-).

We analyze the three different input sizes and report results for Coffeelake (above) using 8 cores and the GPU and Kabylake (below) using 4 cores and the GPU.

As we can see in Figure 3.10, the optimal ratioGPU or partition of parallel iterations at which each implementation achieves the higher throughput depends on the version, input size and platform. In any case, lower or higher ratios degrade performance due to load unbalance, either because the GPU or the CPU multicore are stalled waiting for the other device. As expected, the Atomic version tends to be slower than the NonAtomic one, in particular when the input size is smaller, due to a higher probability of contention in the atomic accesses. This effect is particularly evident when the ratioGPU increases beyond the optimal, meaning that a higher number of diagonals has been offloaded to the GPU, which in turn increases the likelihood of contention in the resulting protected data.

Figure 3.11 shows the throughput (elements per millisecond) that NonAtomic (solid lines) vs Atomic (dashed lines) implementations achieve for the *Dynamic* scheduler. Now we explore what is the GPU chunk size (gpuChunksize) that dynamically offloaded to the GPU achieves the higher throughput.

Let's recall that this scheduler performs many partitions of the iteration space. For instance, a gpuChunksize=2^{10} means that the GPU is repeatedly fed with this chunk size of iterations until we compute the whole iteration space. Again, we study different input sizes and report results for Coffeelake (above) and Kabylake (below) having the GPU along with all the cores working in parallel.

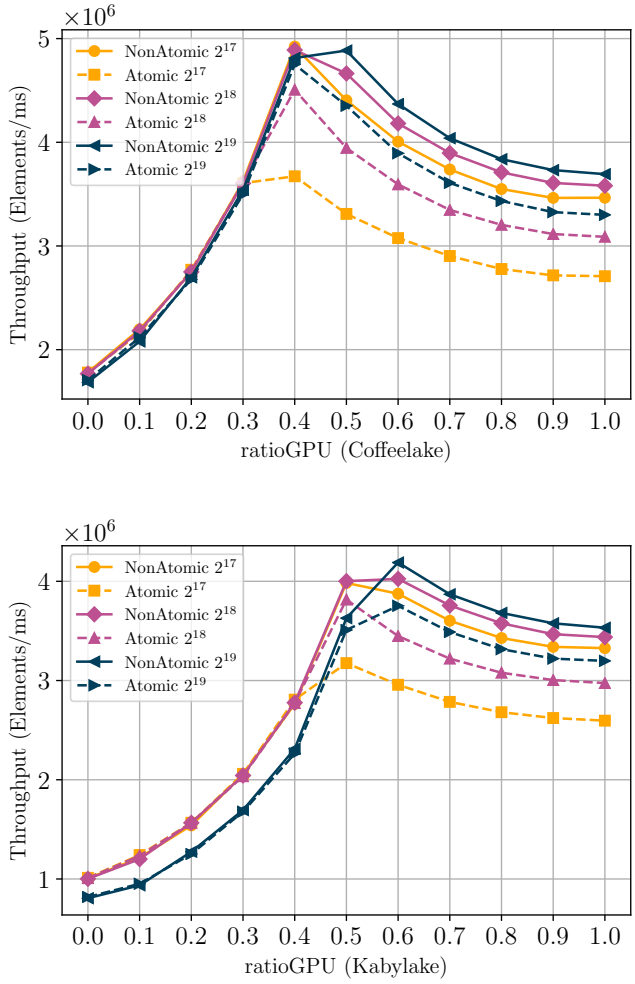


Figure 3.10: Exploring *Static* scheduler in CPU+GPU. X-axes represent the percentage of parallel iterations offloaded to the GPU (ratioGPU).

From Figure 3.11, we find that Coffeelake and Kabylake show a similar behavior. Small chunk sizes degrade throughput due to under utilization of the GPU resources. Too big chunk sizes also degrade throughput now due to load unbalance with the CPU multi-core and due to an increment in the pressure on the FSB (Front Side Bus) that results from the irregular memory access pattern of

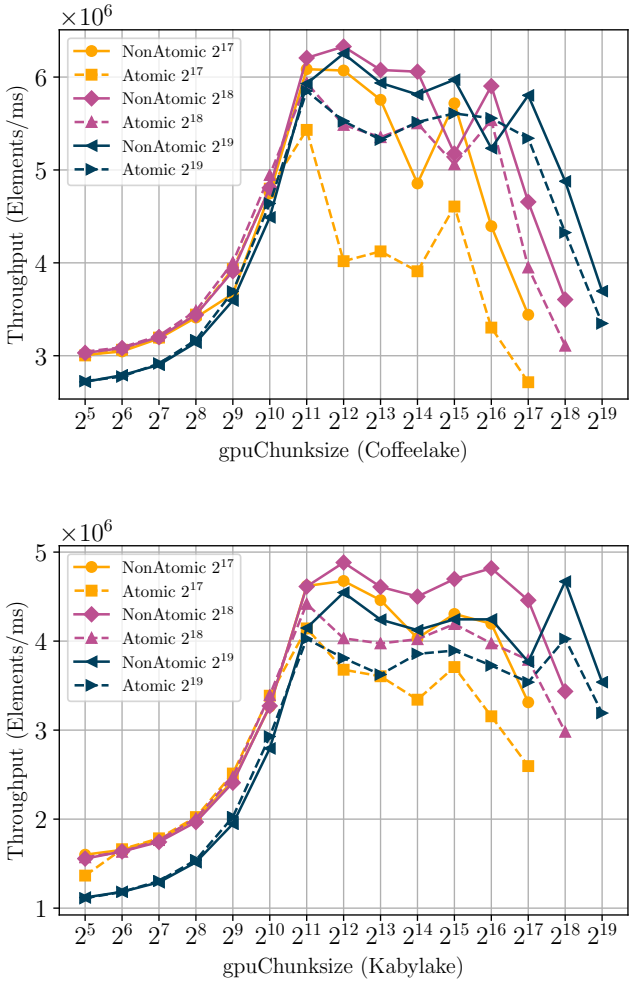


Figure 3.11: Exploring *Dynamic* scheduler in CPU+GPU. X-axes represent the chunk size of parallel iterations offloaded to the GPU.

HetMP. Interestingly, gpuChunksize= 2^{12} (4096 diagonals) achieves the optimal throughput for the NonAtomic version, independently of the input size. In fact, the reported throughput values for the three input data are similar. On the other hand, gpuChunksize= 2^{11} (2048 diagonals) provides the optimal throughput for the Atomic version, independently of the input size. Now, in the Atomic version,

the reported throughput for the smaller input data set tends to be worst than for the bigger input data. Again, this is explained due to a higher probability of contention in the atomic accesses when the input data is small. This issue also explains why the optimal chunk size of the Atomic version is smaller: higher chunk sizes increase the probability of contention among the diagonals being processed in-flight on the GPU.

Both *Static* and *Dynamic* schedulers require offline training to find the optimal ratioGPU or gpuChunksize. In fact, for this running example, we need to execute 11 runs for Static, where we move the ratioGPU from 0 to 1 with 0.1 steps, and 15 runs for Dynamic, while varying the gpuChunksize from 2^5 to 2^{19}. We next explore the behavior of *LogFit* that finds, at runtime, the optimal GPU and CPU chunk sizes and adaptively changes them throughout the computation.

Figure 3.12 shows the evolution of the throughput (elements per millisecond, blue lines) and GPU chunk size (purple lines) that NonAtomic vs Atomic implementations achieve for the *LogFit* scheduler. Here, we show the results for the input time series with 2^{19} elements and report results for Coffeelake (above) and Kabylake (below).

As we see in Figure 3.12, *LogFit* presents the same trends in Coffeelake and Kabelake. First, as expected, the NonAtomic version achieves slightly higher throughput than the Atomic one. Second, the NonAtomic version tends to present noticeable fluctuations of throughput, while the Atomic one shows a very stable result. This is due to how *LogFit* works when looking for the optimal throughput: in the NonAtomic case, after the Exploration Phase, the scheduler finds a chunk size around 2^{12} (as we manually found with *Dynamic*) and enters in the Stable Phase, where it adapts with smooth chunk size variations to changes in the application throughput; on the contrary, in the Atomic case, *LogFit* never exits the Exploration Phase, because the condition that ensures that the throughput has been stabilized is never met. As a consequence, *LogFit* is always exploring bigger and bigger chunk sizes until the iteration space is completely processed. With big chunk sizes it is unlikely to produce significant variations of throughput.

3.3.3.3. Performance vs. Energy analysis

Once we have studied in detail how the different schedulers behave for both NonAtomic and Atomic versions in CPU+GPU executions, we compare the performance and energy efficiency of our implementations, assuming that we have chosen the ratioGPU and gpuChunksize that achieve the best throughput, for both *Static* and *Dynamic*, respectively. We use 8 cores and 4 cores on Coffeelake

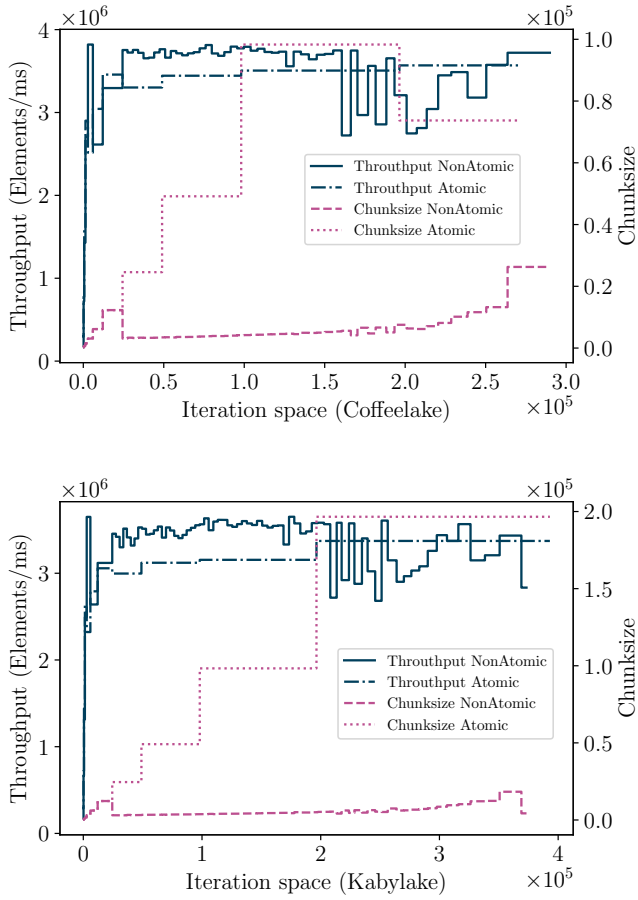


Figure 3.12: Evolution of *LogFit* scheduler in CPU+GPU for input 2^{19}.

and Kabylake, respectively, and the 2^{19} time series, but similar conclusions can be obtained from the other inputs too. These results are shown in Figures 3.13 and 3.14. For reference, we also include results for CPU only and GPU only executions. Table 3.3 contains a summary of results for the different implementations of the Atomic version in Coffeelake and Kabylake.

Figure 3.13 shows that heterogeneous CPU+GPU executions always perform better than any of the homogeneous ones -CPU only or GPU only- and, as already discussed in the previous section, NonAtomic versions perform better than

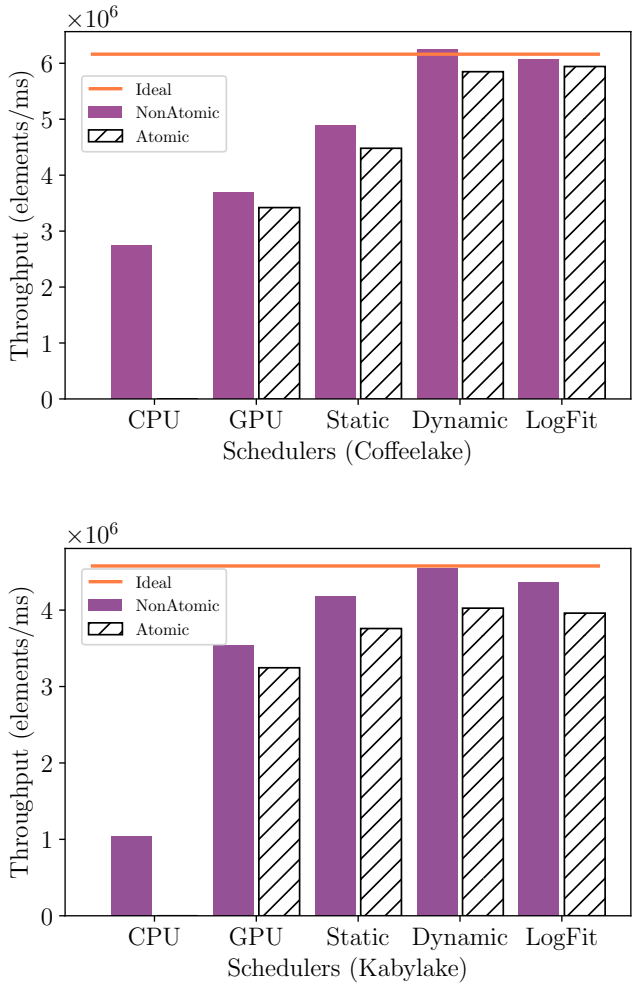


Figure 3.13: Performance comparison: best NonAtomic (solid bar) vs best Atomic (pattern bar) for *Static*, *Dynamic* and *LogFit* schedulers in CPU+GPU runs. The solid line represents ideal performance.

Atomic in all cases. The solid line represents an "Ideal" throughput estimated as the aggregation of the CPU only throughput and NonAtomic GPU only throughput. This ideal throughput does not take into account the overheads due to load unbalance and atomic contention, so it gives us a theoretical upper bound of

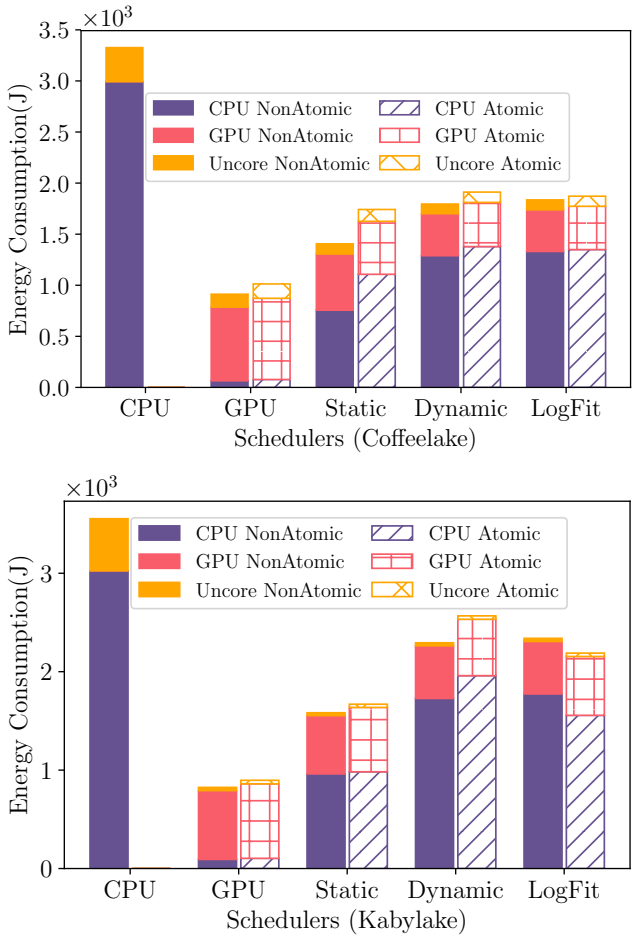


Figure 3.14: Energy breakdown for best NonAtomic (solid bar) and best Atomic (pattern bar) versions *Static*, *Dynamic* and *LogFit* schedulers in CPU+GPU runs.

how efficient the heterogeneous implementations are. As we see in the figure, *Dynamic* is the most performing implementation for NonAtomic versions, while *LogFit* tends to be a good candidate for the Atomic version. This version is only 2.3% slower than NonAtomic in Coffelake and 10.3% in Kabylake. With respect to the Ideal Throughput, in Table 3.3 we see that *LogFit* only presents an overhead of 3.5% and 13.5% on Coffeelake and Kabylake, respectively. Let's

recall that *Dynamic* requires offline training to find the optimal chunk size, while *LogFit* computes the optimal chunk automatically without user intervention.

Table 3.3 also shows the speedup obtained by each version with respect to serial execution on a single CPU core. CPU only version yields almost perfect scalability (7.49x and 3.99x on Coffeelake and Kabylake respectively). GPU only results in 9.8x/8.8x (Coffeelake/Kabylake), but the simultaneous exploitation of both devices improves the performance to up to 15.78x/10.96x (Coffeelake/Kabylake).

Table 3.3: Summary of performance and energy efficiency of Atomic versions on Coffeelake and Kabylake. In bold the optimal implementation for each criterion.

Coffeelake	**CPU 8 cores**	**GPU Only**	**Static**	**Dynamic**	**Logfit**
Throughput (Elements/ms)	2741990	3420660	4480770	5849570	**5942580**
%Diff Ideal Throughput (CPU+GPU)	-55.50%	-44.5%	-27.9%	-5%	**-3.5%**
Speedup (w.r.t. 1 Core)	7.49x	9.82x	11.90x	15.54x	**15.78x**
Energy efficiency (Elements/mJ)	41121	**135147**	116796	71596	73099
%Diff Best Energy	228.3%	0.0%	**71.9%**	88.7%	84.8%
Kabylake	**CPU 4 cores**	**GPU Only**	**Static**	**Dynamic**	**Logfit**
Throughput (Elements/ms)	1041820	3245590	3759050	**4025530**	3960490
%Diff Ideal Throughput (CPU+GPU)	-77.2%	-29.1%	-17.9%	**-12%**	-13.5%
Speedup (w.r.t. 1 Core)	3.97x	8.84x	10.24x	**10.96x**	10.79x
Energy efficiency (Elements/mJ)	38483	**152643**	81496	53279	62508
%Diff Best Energy	296.4%	0.0%	**86.2%**	186.3%	144.1%

From the energy consumption point of view, Figure 3.14 indicates that heterogeneous CPU+GPU executions consume more energy than GPU only, both for NonAtomic and Atomic versions. In particular, Atomic versions are less energy efficient than NonAtomic ones, due to taking longer. On the other hand, although

the heterogeneous CPU+GPU implementations are faster than GPU only ones, they require of an important energy component that is negligible in GPU only case: the consumption due to the CPU cores, which can account for more than 70% of the total energy. As a result, the GPU only implementation is the optimal solution in terms of energy consumption. The best CPU+GPU heterogeneous implementation is *Static*, that for the Atomic version "only" consumes 71.9% and 86.2% more energy than GPU only, for Coffeelake and Kabylake, respectively, as we see in Table 3.3.

Table 3.4: Performance comparison with Matrix profile implementations using Intel Xeon Phi processors.

	Time Series Length			
Throughput (Elements/ms)	2^{17}	2^{18}	2^{19}	2^{20}
HetMP	$6.32 \cdot 10^6$	$6.41 \cdot 10^6$	$6.28 \cdot 10^6$	$5.92 \cdot 10^6$
Intel Xeon Phi 7290	-	-	-	$1.1 \cdot 10^7$
Intel Xeon Phi 7210	$9.34 \cdot 10^6$	$1.07 \cdot 10^7$	$1.06 \cdot 10^7$	$1.01 \cdot 10^7$
Performance / Watt (Throughput/Watio)	2^{17}	2^{18}	2^{19}	2^{20}
HetMP	$6.65 \cdot 10^4$	$6.75 \cdot 10^4$	$6.62 \cdot 10^4$	$6.23 \cdot 10^4$
Xeon Phi 7290	-	-	-	$4.48 \cdot 10^4$
Xeon Phi 7210	$4.34 \cdot 10^4$	$4.99 \cdot 10^4$	$4.98 \cdot 10^4$	$4.70 \cdot 10^4$

Two of the most recent works on discords discovery revolve around parallel optimizations for the Intel Xeon Phi 7290 [72] and Intel Xeon Phi 7210 [71]. For the interested reader, we have compared in table 3.4 the best HetMP results on our Coffeelake (8 threads+GPU, 95W TDP) with the results of [71] on a Xeon Phi 7210 (256 threads, 215W TDP) and [72] on a Xeon Phi 7290 (288 threads, 245W TDP). We compare two metrics: Throughput (Elements per milisecond) as performance of the execution and Energy Performance of the execution (Throughput per Watt). For a time series with 2^{20} elements, HetMP is 41.44% and 46.18% slower than [71] and [72], respectively. However, considering the maximum power of the different architectures, HetMP delivers 32.53% and 38.81% more performance per Watt than [71] and [72], respectively.

3.3.3.4. Summary of results

As a summary, our analysis of performance and energy efficiency for heterogeneous CPU+GPU implementations of HetMP has taught us:

- If accuracy is not an issue, NonAtomic versions perform best than Atomic. In this case, the heterogeneous *Dynamic* scheduler can achieve the fastest execution time. However, the user has to find the optimal GPU chunk size (offline training).
- If accuracy is a requirement, then the heterogeneous *LogFit* scheduler usually provides the fastest or near-fastest execution (only 2.3% slower than NonAtomic *LogFit* in Coffelake and 10.3% in Kabylake). As an additional benefit, this scheduler automatically looks for the optimal GPU chunk size without user intervention.
- If energy consumption is the target criterion, then GPU only execution (both for NonAtomic and Atomic versions) is the implementation of choice.

3.4. Conclusions

The discovery of time series motifs and discords is considered a paramount and challenging problem regarding time series analysis. In this chapter we present HetMP, a heterogeneous implementation of matrix profile algorithm that excels at finding relevant subsequences in time series. We propose and evaluate several static, dynamic and adaptive partition strategies targeting commodity processors, both on homogeneous (CPU multi-core) and heterogeneous (CPU+GPU) architectures. For the CPU+GPU implementation, we explore a heterogeneous parallel_reduce pattern that computes part of the computation onto an OpenCL capable GPU whereas the CPU cores take care of the other part. Our heterogeneous scheduler, built on top of TBB, pays special attention to appropriately balance the computational load among the GPU and CPU cores. The experimental results show that our homogeneous implementation scales linearly and that our heterogeneous implementation allows us to reach near ideal performance on commodity processors that feature an on-chip GPU.

Summarizing, the main contributions of this chapter are:

- We experimentally compare two parallel implementations of the matrix profile computation for multi-core architectures: an ideal static distribution

and a dynamic one based on work stealing (TBB) for multi-core architectures. Our results shows that the dynamic distribution based on TBB suites better for heterogeneous CPU + GPU platforms.

- We extend a previously developed heterogeneous parallel_for template to implement heterogeneous parallel_reduce computations. The reduction phase between the work offloaded to the GPU and the one computed on the CPU cores takes a negligible amount of time and is performed on the CPU. Shared virtual memory, SVM, is leveraged to minimize the CPU-GPU communication overheads.

- We propose two OpenCL kernel implementations for the matrix profile computation on the GPU, based on the reduction operations. The former uses OpenCL atomic operations to ensure accurate reductions, resulting in slower performance. The latter avoids OpenCL atomic operations for a faster performance, resulting in accuracy losses. Both alternatives are compared in terms of performance or accuracy of results with real datasets.

- We evaluate three heterogeneous schedulers (*Static*, *Dynamic* and *LogFit*) from the state-of-the-art on our heterogeneous implementation. Each one requires a different input from the user to control the work offloaded to the GPU. *Static* needs the percentage of work and *Dynamic* the size of the chunk of iterations. In contrast, *LogFit* automatically calculates the working granularity (and offload ratio) required to fully exploit the available devices in the system.

4 Time series on Heterogeneous CPU + FPGA processors

Time series analysis is becoming a major tool on many different domains, such as cloud computing monitoring [48], climate forecasting [52] or earthquake detection [58], among others. A very valuable outcome of these kind of analysis is the discovery of motifs (similarities) or discords (anomalies). Recently, the matrix profile computation has been proposed to accurately and efficiently find motifs and discords [66]. This algorithm consists in cross comparing all subsequences of the time series and recording a score in a resulting time series named "matrix profile". A simple inspection of the maximum and minimum values of the matrix profile is enough to identify the discords and motifs, respectively. To compute the matrix profile, different classes have been incrementally proposed: *STAMP* [66], *STOMP* [20], *SCRIMP* [21], and the latest and most efficient one *SCAMP* [67]. *SCRIMP* has been implemented for different parallel architectures: (1) distributed-memory computers [158]; (2) Intel Xeon Phi KNL processors that integrate 3D-stacked high-bandwidth memory (HBM) [159, 160]; or (3) Heterogeneous CPU + GPU architectures as one of the contributions of this work in the previous chapter 3. However, the current state-of-the-art algorithm to efficiently compute the matrix profile is *SCAMP*, which has been only implemented on a Multi-GPU Cluster [67].

In this chapter, we propose an efficient implementation of *SCAMP* on a heterogeneous platform featuring a multi-core CPU and a High Performance FPGA with integrated High Bandwidth Memory, HBM. This implementation poses some interesting challenges apart from tuning the Matrix profile algorithm to efficiently

run on the FPGA. For instance, several Matrix Profile kernels can be deployed on the FPGA as replicated IPs (FPGA compute units). In order to feed the CPU cores and the FPGA IPs with the corresponding chunks of parallel iterations that guarantee optimal throughput while ensuring load balance, a hierarchical scheduler is proposed. It first partitions the work between the CPU cores and FPGA, and then it proceeds to partition the FPGA work among the different IPs. The system-level (inter-device) scheduler, called *Fastfit*, has been devised to quickly identify the granularity of the work that has to be offloaded to the FPGA in order to achieve both high FPGA utilization and CPU+FPGA load balance. The device-level (intra-device) scheduler is aware of the geometry of the Matrix Profile algorithm (a triangular matrix walk) to also distribute the work evenly among the FPGA IPs. Since the testbed FPGA features 32 HBM banks, we also contribute with a methodology to set the minimum number of active banks that ensure the maximum aggregated memory bandwidth while reducing power consumption. This methodology is based on a model of the HBM bandwidth usage and sharing of banks among IPs. We experimentally validate our scheduler in terms of performance and energy consumption and compare it with previous related and state-of-the-art heterogeneous schedulers.

The rest of the chapter is organized as follows. Section 4.1 describes the Matrix Profile algorithm and the optimizations that we propose for our heterogeneous CPU+FPGA platform. Next two sections describe the *Fastfit* heterogeneous scheduler and HBM analytical model. The experimental results are presented in Section 4.4. The chapter wraps up with conclusions (Section 4.5).

4.1. FPGA-oriented Matrix Profile optimizations

The introduction of time series notation, the matrix profile and distance matrix was detailed in Section 3.1. This section tackles the optimizations *SCAMP* makes over *SCRIMP* to improve performance and the FPGA-oriented optimizations developed. One of the first optimizations is the calculation of the z-normalized Euclidean distance computation from Equation 3.1 using Pearson correlation.

Following the notation of Section 3.1, the z-normalized Euclidean distance, $d_{i,j}$, between subsequences $T_{i,m}$ and $T_{j,m}$ can be efficiently computed using the following equations:

$$df_1 = 0; \quad df_i = \frac{t_{i+m-1} - t_{i-1}}{2} \tag{4.1}$$

$$dg_1 = 0; \quad dg_i = (t_{i+m-1} - \mu_i) + (t_{i-1} - \mu_{i-1}) \tag{4.2}$$

$$Cov_{1,j} = \sum_{k=0}^{m-1} (t_{1+k} - \mu_1)(t_{j+k} - \mu_j) \tag{4.3}$$

$$Cov_{i,j} = Cov_{i-1,j-1} + df_i \cdot dg_j + df_j \cdot dg_i \tag{4.4}$$

$$norm_i = \frac{1}{||T_{i,m} - \mu_i||} \tag{4.5}$$

$$P_{i,j} = Cov_{i,j} \cdot norm_i \cdot norm_j \tag{4.6}$$

$$d_{i,j} = \sqrt{2 \cdot m \cdot (1 - P_{i,j})} \tag{4.7}$$

where μ_i is the mean of $T_{i,m}$. Basically, the distance between subsequences i and j is computed in Equation 4.7 using the Pearson correlation. In turn, Pearson is obtained in Equation 4.6 from the covariance of the pair of subsequences (Equation 4.4) and the norms of both subsequences (Equation 4.5). Equation 4.4 computes the non-scaled covariance for the range of indexes $2 \leq i \leq n, i < j \leq n$ based on the initialization performed in Equation 4.3. Note that indexes i and j start at 1, as done in related works [66, 67] for the sake of simplifying notation.

Algorithm 2 shows the Matrix Profile sequential implementation of the matrix profile computation using Pearson correlation. In lines 1-3 we initialize n (length of T), the matrix profile, MP, the matrix profile index, I and the set of *Diagonals* that must be traversed. In line 4, we pre-compute $df_i, dg_i, norm_i$ and $Cov_{1,j}$, for Exclusion Zone, $EZ < i < n$, as described in the appendix, which are stored in their corresponding vectors to avoid repeating unnecessary computations. The outer loop (line 5) traverses the diagonals whereas the inner loop (line 6) processes each element of the diagonal in $O(1)$. The values $Cov_{i,j}$, $P_{i,j}$ and $d_{i,j}$ are not stored, but computed as C, P and d and later discarded after they are used to update MP and I in lines 12-13. These two checks of lines 12-13 are needed because although we traverse only the upper-triangular submatrix (because $d_{i,j} == d_{j,i}$), $MP_i = min(D_i)$ and I_i can be different to $MP_j = min(D_j)$ and I_j respectively.

For the FPGA and heterogeneous implementation, some relevant optimizations were applied to this algorithm:

- Instead of comparing with $d_{i,j}$ looking for the $min(D_i)$, we can just look for the largest $P_{i,j}$. That way we save some floating point operations and a square root. Now, the MP vector temporary stores Pearson correlations

Algorithm 2: The Matrix Profile algorithm using Pearson Correlation

Input: A time series T, m and EZ
Output: Matrix Profile MP, Matrix Profile Index I

1 $n \leftarrow Length(T)$
2 $MP \leftarrow \infty, I \leftarrow \texttt{zeros}$
3 $Diagonals \leftarrow (EZ + 1 : n - m + 1)$
4 $df_i, dg_i, norm_i, Cov_{1,j} \leftarrow preCompute(T, m, EZ)$
5 **for** k ***in*** *Diagonals* **do**
6 **for** $i \leftarrow 1$ ***to*** *length(k)* **do**
7 $j \leftarrow i + k - 1$
8 **if** $i == 1$ **then** $C \leftarrow Cov_{1,k}$
9 **else** $C \leftarrow C + df_i \cdot dg_j + df_j \cdot dg_i$
10 $P \leftarrow C \cdot norm_i \cdot norm_j$
11 $d \leftarrow \sqrt{2 \cdot m \cdot (1 - P)}$
12 **if** $d < MP_i$ **then** $MP_i \leftarrow d, I_i \leftarrow j$
13 **if** $d < MP_j$ **then** $MP_j \leftarrow d, I_j \leftarrow i$
14 **return** ***MP, I***

and at the end of computation motifs can be identified by the largest values, and discords by the smallest ones. If we really need the distance values, a quick traversal of MP applying Equation 4.7 produces the Matrix Profile as we have defined in the previous section.

- The FPGA architecture excels at regular computations with the simplest control flow. We can remove the conditional expression (line 8 in Algorithm 2) using loop peeling, this is, unwinding the first iteration from the loop. The host (CPU) can take care of this first iteration and correspondingly update MP and I. This first iteration consumes less than 0.001% of the total execution time in our experiments. As a result, we save FPGA resources which translates into more Matrix Profile kernels (IPs) fitting into the FPGA fabric.

- Our heterogeneous implementation of Matrix Profile for CPU+FPGA platforms splits the *Diagonals* parallel iteration space in chunks of diagonals. Each thread (one per CPU core) and each FPGA IP (or FPGA kernel) can process different chunks in parallel. Each non-overlapping chunk of diagonals is identified by the range $[begin, end)$. Each CPU thread has a private copy of MP and I using TBB's combinable class that provides thread-local storage and a user-friendly reduction method. Each FPGA IP also has a

private copy of MP and I, now using the High Bandwidth Memory banks available in our FPGA device. The required reduction phase that results in the final MP and I, consumes less than 0.09% of the total execution time, according to our experiments. The implementation details of the reduction operation are explained in Chapter 3, although in that Chapter we targeted a CPU+GPU platform and here we can have up to 40 FPGA IPs instead of a single GPU.

Algorithm 3 shows the pseudocode of the host code, that basically takes care of the initialization[1] (lines 1-4 in Algorithm 2) and then it precomputes the first iteration of the i-loop for all the diagonals. These computations are run sequentially but in our experiments the worst case consumes only 0.13% of the total execution time. The other 99.87% of the time is consumed in the `heterogeneous_parallel_for` call provided by our HBB library (Heterogeneous Building Blocks) [146] that we describe in section 4.2.

Algorithm 3: Matrix Profile Host

```
1  n ← Length(T)
2  MP ← −∞, I ← zeros
3  Diagonals ← (EZ + 1 : n − m + 1)
4  df_i, dg_i, norm_i, Cov_{1,j} ← preCompute(T, m, EZ)
5  for k in Diagonals do
6  |   P ← Cov_{1,k} · norm_1 · norm_k
7  |   if P > MP_1 then MP_1 ← P, I_1 ← k
8  |   if P > MP_k then MP_k ← P, I_k ← 1
9  heterogenous_parallel_for(Diagonals, Body)
10 return MP, I
```

As we describe later, this `heterogeneous_parallel_for` function requires a `Body` object that, among other things, encapsulates how to process a chunk of iterations (diagonals in this case) on the CPU and on the accelerator. Algorithm 4 shows the FPGA kernel implementation that takes cares of a chunk of diagonals in the range $[begin, end)$. Each FPGA kernel receives the initialized variables and writes in private MP and I arrays executing the i-loop starting at iteration $i = 2$. Note that we now compute and store the Pearson correlation instead of the distance.

[1] Note that in contrast to Algorithm 2, now MP is initialized with $-\infty$ because it now stores Pearson correlations instead of distances.

Algorithm 4: Matrix Profile FPGA Kernel

Input: MP, I, Cov, df, dg, norm, begin, end
Output: MP and I

```
1 for k ← begin to end − 1 do
2 |   C ← Cov_{1,k}
3 |   for i ← 2 to length(k) do
4 |   |   j ← i + k − 1
5 |   |   C ← C + df_i · dg_j + df_j · dg_i
6 |   |   P ← C · norm_i · norm_j
7 |   |   if P > MP_i then MP_i ← P, I_i ← j
8 |   |   if P > MP_j then MP_j ← P, I_j ← i
9 return MP, I
```

The FPGA kernel is implemented in OpenCL and compiled into an FPGA bitstream using the Intel `aoc` compiler. As recommended in the FPGA optimization guide [161], we follow a single-task approach (also known as single work-item), in which the OpenCL kernel resembles a sequential C implementation. For these type of kernels the OpenCL NDRange[2] is set to (1, 1, 1), so a single thread is invoked on each FPGA IP. This results in loop pipelining and overlapping of data transfers and computations between loop iterations.

However, as we can see in Figure 4.1, the inner i-loop that traverses a diagonal exhibits a loop carried dependence because iterations i and i' can RMW (read-modify-write) the same positions in MP and I. This dependence prevents the pipeline implementation of the loop and results in a highly inefficient FPGA execution.

However, a closer look at the loop body reveals that such a potential RMW conflict can be avoided in our implementation. Figure 4.1 shows a simplification of the pipeline execution of the i-loop, where IL is the *Issue Latency* (a.k.a. Initiation Interval) or number of clock cycles between consecutive loop iterations. We also show CL, *Completion Latency* that we use in section 4.2.2 to model the kernel throughput. The figure also shows the potentially conflicting statements D and E accessing the same MP position. Since for a given diagonal k, index j walks through $j = i + k - 1$, two iterations i and $i' = i + k - 1$ can RMW the same position of MP: MP_j in E and $MP_{i'}$ in D. In the figure we simplify the situation assuming that $IL = 1$ and statements D and E require a clock cycle, but in general IL can be higher and the statements may require c cycles.

[2] In the OpenCL standard, the NDRange represents the 3D space of parallel iterations.

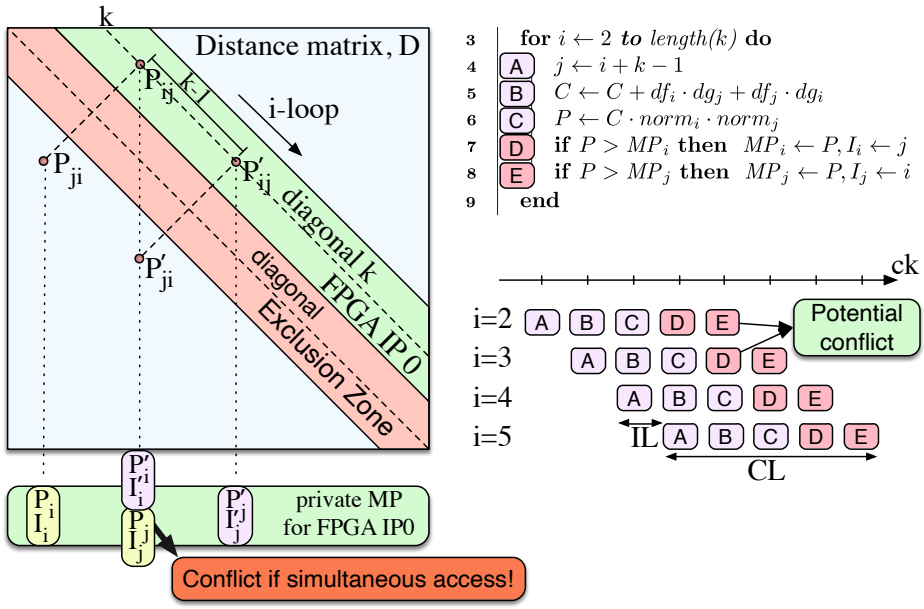


Figure 4.1: Potential conflict due to loop-carried dependence in the inner loop and pipeline execution in the FPGA.

Therefore, if we can assure that statement E of iteration i finishes before statement D of i', then the loop can be safely pipelined. Without loss of generality, if statement E of iteration i access MP_j in the interval of cycles $[t, t+c)$, statement D of iteration i' access the same position in the interval $[t + IL \cdot (k-1) - c, t + IL \cdot (k-1))$, but these two intervals do not overlap if $t + c < t + IL \cdot (k-1) - c$. In other words, if $c < (IL \cdot (k-1))/2$.

In our algorithm, the first diagonal (the smallest k) is $k = EZ + 1$, and EZ is 256 as recommended in the literature [21, 67]. On the other hand, the `aoc` compiler reports $IL \approx 6$, so the number of cycles, c, required to compute statements D and E should be smaller than 768 cycles. This is actually the case considering that each of these statements only includes a read operation from HBM, a comparison and two writes in HBM memory. If a smaller EZ is advised, we can always offload to the FPGA only the diagonals that are far enough from the main diagonal.

Therefore, as we know that there are not loop carried dependencies in the i-loop, we force the pipelining implementation with the `Pragma("ivdep")` di-

rective just before the loop. Additionally, the FPGA kernels have been compiled with `-fp-relaxed -fpc`, that according to the FPGA OpenCL programming guide [162], result in floating-point optimizations including balanced tree hardware and elimination of intermediary rounding operations.

4.1.1. Initial performance assessment

As detailed at the beginning of Chapter 4, *SCAMP* is the state-of-the-art for Matrix Profile computation, ahead of *SCRIMP*, the previous one. However, *SCAMP* has been only implemented in CPU and GPU, with no previous results on FPGA architectures. Performance on an FPGA is highly dependent of the kernel operations implemented. Hence, it is necessary to verify that *SCAMP* is the most suitable implementation for matrix profile computation on FPGA as well. Both alternatives, *SCAMP* and *SCRIMP*, have been implemented and executed on the FPGA comparing their initial performances.

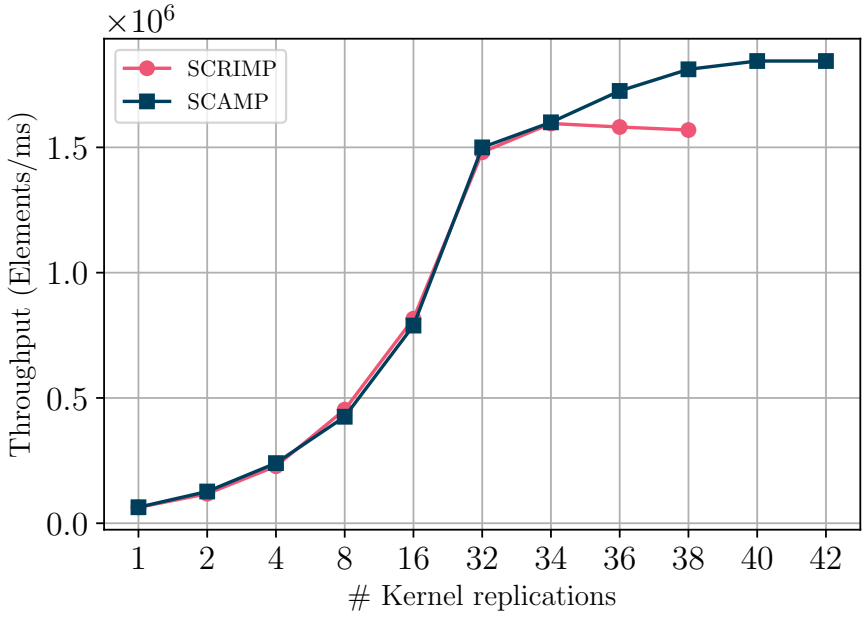


Figure 4.2: Exploring the performance in SCRIMP and SCAMP only-FPGA executions, for different number of kernel replications, N_{IP}, and a time-series with 2^{17} elements.

Figure 4.2 shows the performance results of *SCRIMP* and *SCAMP* executions for different number of kernel replications (IPs), one of the FPGA optimizations described in Section 4.1. The performance is measured using the Throughput, the number of elements computed per millisecond. The result shows that *SCAMP* improves *SCRIMP* in a 15.62% comparing their maximum performances. Published results of CPU and GPU implementations of *SCAMP* and *SCRIMP* [21, 22] show that *SCAMP* is faster because the first one uses less operations than *SCRIMP* in order to compute the subsequences similarity. This is, *SCAMP* is more efficient algorithmically: the Pearson correlation requires only 2 FP operations (2 multiplications) while the euclidean distance requires 9 operations (6 multiplications, 1 division and 2 subtractions). Although counter-intuitive, this difference in the number of FP operations is not relevant in our FPGA implementations in which an Initiation Interval (or IL in Figure 4.1) of 1 is achieved both for the *SCAMP* and *SCRIMP* implementations. In other words, *SCRIMP* has a larger latency per iteration, but it exhibits the same throughput than *SCAMP*, which render both implementation equivalents for a large number of iterations. Hence, as expected, for the same number of IPs both implementations have the same performance, as can be seen in Figure 4.2.

However, the euclidean distance requires more FP operations which translates in more hardware in the FPGA. That way, our FPGA can accommodate less replicas of the *SCRIMP* IP (up to 38) than of the simpler *SCAMP* IP (up to 42). Besides, since the number of replicas have an impact on the FPGA frequency (due to placement and routing issues), *SCRIMP* maximum throughput is achieved for 34 IPs, whereas *SCAMP* keeps improving performance until 42 IPs, as can be seen in Figure 4.2. These difference in the kernel replication is what gives *SCAMP* up to 15.62% of improvement in performance compare to *SCRIMP* in our experiments.

4.2. Fastfit: Hierarchical Heterogeneous Scheduler

4.2.1. Scheduling engine

Our scheduler is based on the Heterogeneous Building Blocks (**HBB**) library [146]. It is a C++ template library based on TBB, which takes advantage of heterogeneous processors and facilitates their usage and configuration. HBB aims to make easier the programming for heterogeneous processors by automatically partitioning and scheduling the workload among the CPU cores and OpenCL capable accelerators. HBB relies on OpenCL as the accelerator back-end for the

sake of availability, portability, and programmability features, but the scheduling framework and policies of HBB could be easily adapted to other programming models or high level synthesis tools. Our library (HBB) offers an abstraction layer that hides the initialization and management details of TBB and OpenCL constructs (contexts, command queues, device_ids, etc), thus the user can focus on his own application logic instead of dealing with thread management and synchronizations. The current version offers a `heterogeneous_parallel_for()` function template to run on heterogeneous CPU-GPU and CPU-FPGA systems.

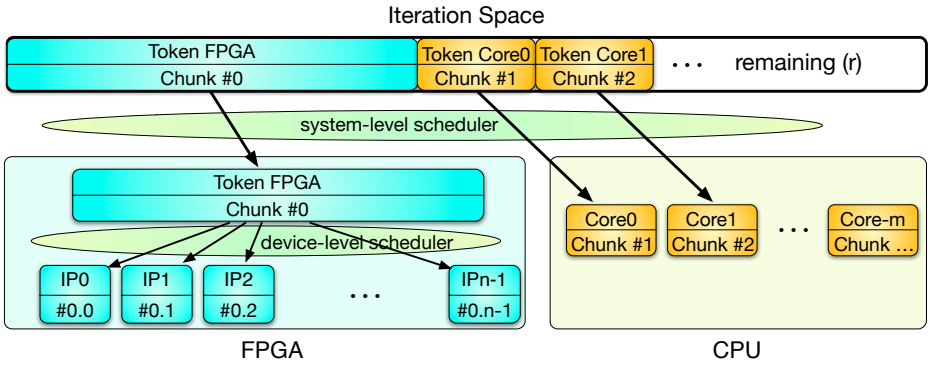


Figure 4.3: System and device level schedulers used to partition the iteration space.

Figure 4.3 illustrates how the proposed hierarchical heterogeneous scheduler works. The system-level scheduler offloads chunks of iterations to the FPGA as soon as the FPGA becomes idle, and also assigns CPU chunks to each core in the CPU. The device-level scheduler takes care of partitioning each FPGA chunk into sub-chunks to appropriately feed each of the FPGA IPs. The iteration space includes: i) chunks of iterations that have already been assigned to the FPGA (blue); ii) chunks of iterations already assigned to the CPUs (orange); and iii) remaining iterations (white).

The system-level scheduler is designed as a two-stage pipeline, Stage1 and Stage2, implemented on top of TBB. Thanks to the pipeline tokens we can easily control when the FPGA or the CPU cores are idle or busy. We initialize the pipeline object with one FPGA token and as many CPU tokens as the number of cores available. The tokens are circulating through the pipeline, being recycled at Stage1 entry once they exit Stage2. Depending on the arriving token, in Stage1 the size of a CPU or FPGA chunk of iterations is computed (as we explain in

Section 4.2.2), and the chunk is extracted from the set of remaining iterations, r. In the parallel Stage2, either the CPU core or the FPGA processes the previously selected chunk.

We also initialize the TBB scheduler with as many worker threads as tokens (# of CPU cores plus 1 –the FPGA–). That way, if we have one FPGA and two CPU cores, three worker threads are able to process three chunks of iterations in parallel. However, the FPGA can have several IPs (FPGA compute units) and therefore processing a chunk on the FPGA involves our device-level scheduler that evenly distributes the FPGA chunk among the available IPs (as described in Section 4.2.3). Note that the worker thread that processes the FPGA chunk/token, is the one: i) running the device-level partitioning; ii) offloading each sub-chunk to each IP; and iii) blocking until all IPs have finished processing the sub-chunk. The over-subscription of the CPU cores is negligible because, although the TBB scheduler has one extra thread (one more than the number of cores), this thread is usually blocked while the FPGA is working. An alternative that we discarded consists in having one worker thread per FPGA IP and CPU core, but this results in too much over-subscription since in our platform we can have 40 IPs and 8 CPU cores, which translates into 48 worker threads. Besides, the FPGA OpenCL driver does not support concurrent offload requests from more than one worker thread.

One of the major advantages of this hierarchical scheduler comes from the fact that the implementation is optimized at two levels to fully exploit the parallel capabilities of the CPU and the FPGA: (1) At a system-level, the `parallel_for()` implementation [163], each compute unit (FPGA or a CPU core) is represented as a token that traverses the pipeline at its own pace. Thus, we avoid unnecessary synchronization points between compute units with different computing performance. (2) At a device-level, the performance of the FPGA is exploited by parallelizing the work assigned to the FPGA and balancing it for its different IPs cores.

In contrast, other state-of-the-art approaches [164, 165] suffer from load unbalance due to the usage of *fork-join* patterns with implicit synchronization points between the CPU and the accelerator.

4.2.2. Fastfit system-level scheduling algorithm

The system-level scheduler works at runtime and is designed as a two-phase strategy consisting of: the *Training Phase*, which finds the near-optimal chunk-sizes for the FPGA and the CPU that optimize the throughput in both devices

while ensuring load balance; and the *Exploitation Phase*, which keeps this peak performance along the iteration space. Algorithm 5 depicts both phases.

A key component of the *Training Phase* is an analytical model that estimates the FPGA throughput, i.e., *elements per ms* computed when executing a chunk of parallel iterations. Our model assumes that an FPGA IP is internally implemented as a pipeline from which it estimates the near-optimal FPGA chunksize that maximizes the FPGA throughput. This model results in a good balance between accuracy and simplicity. The pipeline is characterized by two latencies: issue and completion latencies. The *Issue Latency*, IL, is the number of cycles required between issuing two consecutive independent iterations, which is also know as the *Initiation Interval*. On the other hand, the *Completion Latency*, CL, is the number of cycles until the result of a parallel iteration is available. Both latencies are in most cases sufficient to estimate the execution time of an FPGA kernel: the issue latency represents the time between dispatching two consecutive iterations of the kernel loop, while the completion latency depends on the depth of the pipeline and is the time required to fill it up.

Algorithm 5: Fastfit System-level scheduler

```
// Training Phase
```

Input: Frecuency (F), δ, ρ

1 $t_{C_m}(1), t_{F_m}(1) \leftarrow$ Equation 4.9
2 $t_{F_m}(\delta) \leftarrow$ Equation 4.10
3 $CF \leftarrow$ Equation 4.16 $\leftarrow$ Equation 4.11 & Equation 4.12
4 $CC \leftarrow$ Equation 4.19 $\leftarrow$ Equation 4.17 & Equation 4.18
5 **return** CF, CC

```
// Exploitation Phase
```

Input: CF, r, φ

6 $CF = min(CF, r)$
7 $CC = CF/\varphi$
8 **return** CF, CC

As we show in Algorithm 5, the *Training Phase* only requires to sample the CPU and FPGA throughput running one iteration (a diagonal in our application) on the CPU, and two chunks of iterations on the FPGA and then recording the corresponding execution times. In line 1, the first chunk for the FPGA and the CPU is made of 1 iteration each one. In line 2, the second chunk, only for the FPGA, contains a representative number of parallel iterations δ (in our

study we find that 5% of the iteration space is enough to characterize the FPGA throughput for our application).

Let's suppose that we know the clock frequency of the FPGA (denoted by F and provided by the aocl compiler in a report file). When we offload a chunk of parallel iterations of size CF, to the FPGA, then our model estimates the time to complete them as,

$$t_{F_e}(CF) = (CF \cdot IL + DL) \cdot \frac{1}{F} \tag{4.8}$$

where DL represents the number of cycles required to traverse the pipeline after issuing a parallel iteration. The Completion Latency can be defined as $CL = IL+DL$. By applying Equation 4.8 to the two FPGA chunks of 1 and δ iterations, respectively, we obtain a system of two equations and two unknowns:

$$t_{F_m}(1) = t_{F_e}(1) = (IL + DL) \cdot \frac{1}{F} \tag{4.9}$$

$$t_{F_m}(\delta) = t_{F_e}(\delta) = (\delta \cdot IL + DL) \cdot \frac{1}{F} \tag{4.10}$$

As we know F, δ, $t_{F_m}(1)$ and $t_{F_m}(\delta)$ (lines 1 and 2 of Algorithm 5), we can solve IL and DL as,

$$IL = \frac{t_{F_m}(\delta) - t_{F_m}(1)}{\delta - 1} \cdot F \tag{4.11}$$

$$DL = t_{F_m}(1) \cdot F - IL \tag{4.12}$$

From Equation 4.8 and the previous expressions, we model the *FPGA estimated throughput*, λ_{F_e}, for a chunk CF of parallel iterations as,

$$\lambda_{F_e}(CF) = \frac{F}{IL + DL/CF} \tag{4.13}$$

Peak performance is attained with full pipelines, in which the completion latency is hidden. Latency hiding is achieved by executing a large enough chunk of independent iterations. Ideally, when the DL is completely hidden ($DL/CF \rightarrow 0$), then the issue latency determines the run time and we attain peak performance. From Equation 4.13 we compute the *peak performance* or *ideal throughput*, that we denote $\lambda_{F_{peak}}$ as,

$$\lambda_{F_{peak}} = \frac{F}{IL} \tag{4.14}$$

The goal of the *Training Phase* in our scheduler is to find a sufficiently large chunk of parallel iterations that guarantees that the estimated FPGA throughput is above a certain threshold of the peak performance, $\rho \cdot \lambda_{F_{peak}}$. Typically we seek ρ values in the range $[0.9, 0.99]$, meaning that we aim to look for chunks that achieve throughputs that are within 90% and 99% of the peak performance. From Eqs. 4.13 and 4.14, and for a specified ρ, we know,

$$\frac{F}{IL + DL/CF_\rho} \geq \rho \cdot \frac{F}{IL} \tag{4.15}$$

In other words, the near-optimal chunk size of parallel iterations that guarantee a throughput above a ρ threshold of the peak, CF_ρ, is computed as,

$$CF_\rho \geq \frac{DL}{IL} \cdot \frac{\rho}{1-\rho} \tag{4.16}$$

This steps can be summarized in Line 3 of Algorithm 5 where, using the execution times computed in Lines 1-2, Eqs. 4.11 and 4.12 can be solved. These solutions allow to solve Equation 4.16 to get the near-optimal FPGA chunk size, CF_ρ.

Likewise, we can discover the optimal chunk for each CPU core from the FPGA chunk computed above, as can be seen in Line 4 of Algorithm 5. As input we take the execution time of one parallel iteration in the CPU $t_{C_m}(1)$ -already computed- and the number of elements in the corresponding parallel iteration, N_C. This is, in fact, the number of elements in the corresponding single diagonal of the matrix. Also, we calculate N_F that represents the aggregated number of elements in all the diagonals of chunk CF_ρ. From them, we compute the throughput of the CPU and FPGA for both chunks as can be seen in Eqs. 4.17 and 4.18.

$$\lambda_C = \frac{N_C}{t_{C_m}(1)} \tag{4.17}$$

$$\lambda_F = \frac{N_F}{(CF_\rho \cdot IL + DL)/F} \tag{4.18}$$

With this, the relative speed of the FPGA over the CPU is $\varphi = \frac{\lambda_F}{\lambda_C}$. It is advisable that the FPGA and CPU cores take the same time to compute their

corresponding chunks, which results in the recommended CPU chunk size, CC, which can be approximately computed as:

$$CC = \frac{CF_\rho}{\varphi} \tag{4.19}$$

After the first computation of chunk sizes CF and CC, the scheduler transitions to the *Exploitation Phase* (lines 6-8 in Algorithm 5), where we keep processing chunks of iterations on the CPU cores and FPGA and measuring the actual resulting throughput to update the relative speed among devices, φ. If the number of remaining iterations, r, is large enough, CF is kept as computed in the *Exploitation Phase* but when there are not enough iterations to feed the FPGA with CF iterations, then the remaining, r, are assigned as an FPGA chunk instead (see line 6). CC is recomputed each time to adapt to changes in φ (see line 7).

FPGA is mostly used for regular codes where the workload remains constant throughout the iteration space and, therefore, the throughput. Due to this regularity, in most cases it is unnecessary to spend time controlling and recomputing the optimal chunksize from time to time to ensure the throughput do not decrease, as others schedulers designed for irregular codes does [146].

This is one of the strengths of *Fastfit*. Thanks to a very short *Training Phase* (only two chunk executions for FPGA and one for CPU) *Fastfit* can estimate the chunksize for an optimal throughput for FPGA and CPU and spend most of the execution time in the *Exploitation Phase*. In comparison with *Logfit*, *Fastfit* also avoids the overhead of recomputing the optimal accelerator chunksize and achieves a nearly ideal performance as we will analyze in detail later on in Section 5.4.

4.2.2.1. Fine tuning chunk size for Matrix Profile

In the previous section we have estimated a near-optimal FPGA chunk size, CF_ρ, that delivers an FPGA throughput close to the theoretical maximum. For any problem in which the geometry of the chunk is not an issue or the user does not have additional information about that, the previously computed chunk size can be a reasonable solution for the remainder of the application execution. However, we are aware that our problem exhibits a triangular geometry as can be seen in Figure 4.4, and our goal is, once the optimal chunk size has been found in the *Training Phase*, to guarantee that each new chunk assigned to the FPGA always computes the same workload.

Let's suppose that the first FPGA chunk size found in the *Training Phase*, CF_ρ, traverses diagonals in the range $[x_0, x_1)$, accounting all of them to N_F elements. In the example of Figure 4.4, the next chunk of iterations, CC, is computed on the CPU. Let's assume we transition to the *Exploitation Phase* and that a new FPGA chunk size, CF, has to be computed. Let's note that, in order to keep the desired FPGA throughput, we also have to keep almost constant the workload of all the FPGA chunks. So now, the problem is computing the next CF given that the first diagonal of the new chunk is x_b, so that the number of total elements in this chunk is also N_F.

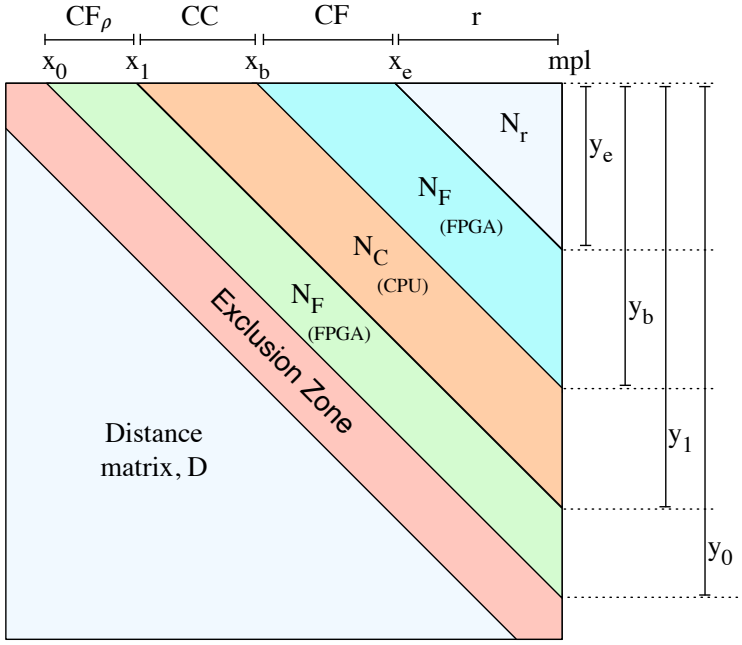


Figure 4.4: Correction in the chunk size due to the triangular geometry of the problem.

The number of iterations/diagonals in CF is $CF = x_e - x_b$, for the new range $[x_b, x_e)$. Note that the index of the last diagonal is $mpl = n - m + 1$ (see Section 4.1). Figure 4.4 shows that $x_i = mpl - y_i + 1$ where y_i is the number of elements in diagonal x_i. Since consecutive diagonals only differ in one element, in the chunk CF_ρ the aggregation of the first and last diagonal is $sum = y_0 + y_1 + 1$. This sum is equal to the aggregation of the adjacent interior diagonals, it is $sum = y_0 - 1 + y_1 + 2$, and so on. This results in,

$$N_F = \frac{CF_\rho}{2} \cdot (y_0 + y_1 + 1) \tag{4.20}$$

Now we want to compute CF and x_e, knowing N_F, x_b and that $CF = x_e - x_b = y_b - y_e$, using the same equation for the CF chunk:

$$N_F = \frac{CF}{2} \cdot (y_b + y_e + 1) = \frac{CF}{2} \cdot (2 \cdot y_b - CF + 1) \tag{4.21}$$

that is a quadratic equation from which we can easily solve CF and later x_e. That way, during the *Exploitation Phase*, we ensure that the number of elements computed on each FPGA chunk remain almost equal and that the FPGA yields an almost constant throughput as we validate in Section 4.4.

In our experiments, when activating this fine tuning of the chunk size in our scheduler we observe a 1% improvement in the performance w.r.t. the not geometrically aware scheduler (the default one).

4.2.3. Fastfit device-level scheduling algorithm

As introduced in Section 4.1, the FPGA can actually include N_{IP} compute units (or IPs) that work in parallel. The goal of the device-level scheduler is to partition each FPGA chunk of CF iterations among the N_{IP} IPs.

A naive distribution that disregard the geometry of our problem, would be the *Block* partition that just distributes the matrix diagonals in equal sub-chunks: $chunkIP = \frac{CF}{N_{\text{IP}}}$. This is the default policy in our scheduler. However, as we validate in Section 4.4, a better approach to perform the partition, which we call *Balanced*, do consider the number of elements in each diagonal.

Starting from Equation 4.21, the N_F elements of the FPGA chunk CF have to be partitioned into N_{IP} sub-chunks, $CF_0, CF_1, \dots CF_{N_{\text{IP}}-1}$, each one with approximately N_F/N_{IP} elements. Therefore we can compute each sub-chunk iteratively by following this expression,

$$\frac{N_F}{N_{IP}} = \frac{CF_i}{2} \cdot (2 \cdot y_{b_i} - CF_i + 1) \quad \forall i \in \{0 \dots N_{\text{IP}} - 1\} \tag{4.22}$$

from which each CF_i can be computed starting with $i = 0$ and $y_{b_0} = y_b$, and updating at each step $y_{b_i} = y_{b_{i-1}} + CF_i$. With this *Balanced* partition strategy each IP gets approximately the same number of elements (± 1 diagonal). In

our experiments this translates into a negligible unbalance among IPs (less than $10^{-5}\%$).

4.3. HBM exploitation

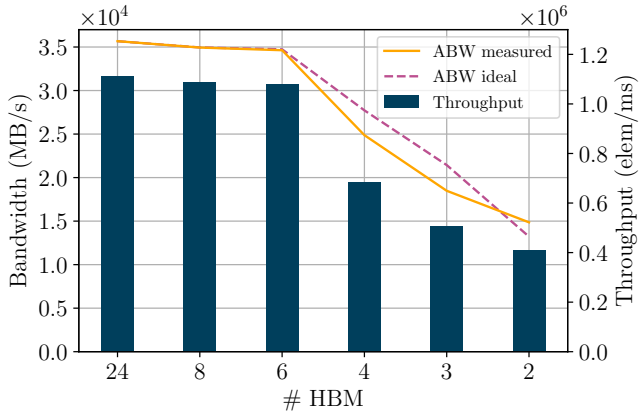
In case there are available resources on the FPGA fabric to instantiate several replicated FPGA IPs, we must tackle the issue of carefully placing and distributing among the HBM banks the data that each one of these IPs accesses, in order to efficiently exploit the memory bandwidth offered by the banks. Previous versions of the FPGA SDK for OpenCL compiler offered an optimization based on the generation of multiple compute units for enabling kernel replication through `pragma` and attribute `__attribute__((num_compute_units()))`. To emulate this feature, we replicate the kernel in the OpenCL source code N_{IP} times using C macros. This way the compiler implements each replicated kernel or FPGA IP as an unique pipeline.

One difference of the kernel replication with respect to the compute units compiler generation is that there is not a hardware scheduler unit built in the FPGA. That is the reason why we implement the device-level scheduler (see Section 4.2.3) responsible for the partition of the FPGA chunk and the dispatch of the corresponding sub-chunks among the different IPs. Another key difference of our kernel replication strategy is that it allows us to control the HBM bank where each IP will access local data. Trivially, one IP can access its local data from one HBM bank. Thus, by increasing the number of replicated IPs, each one accessing data from a different HMB bank, we can achieve higher throughput when exploiting the aggregated memory bandwidth of the concurrent memory banks. However, increasing the number of active memory banks rises power consumption and requires additional FPGA resources to orchestrate all the bank's memory accesses. In case that one IP does not exhaust the available bank bandwidth, then two (or more) IPs could have allocated their data on one HBM bank. This bank sharing solution would optimize the memory bandwidth usage of each HBM bank, reduce the number of active HBMs, and decrease power consumption and FPGA resources, while obtaining the maximum aggregated bandwidth achievable for a given number of replicated IPs.

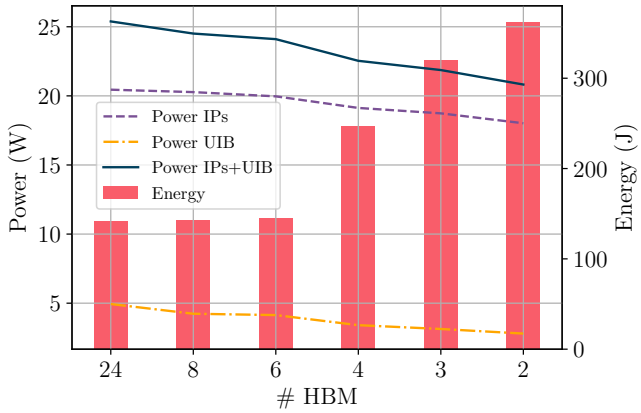
In this section we present a methodology that allows us to: i) select the optimal number of IPs that can access each bank in order to ensure optimal memory bandwidth usage of a HBM bank; and also ii) set the minimum number of active banks that ensure the maximum aggregated memory bandwidth for a given number of IPs. This methodology is based on a model of the HBM

bandwidth usage that we explain next. Figure 4.5 illustrates the accuracy of our model and its applicability.

4.3.1. Modeling bandwidth usage for HBM



(a) Memory Bandwidth and Throughput



(b) Power and Energy

Figure 4.5: Model estimation and performance evaluation for 24 IPs and different number of HBMs: (a) Comparing ideal and measured aggregated memory bandwidth vs application throughput -the higher the better-; (b) Comparing Power dissipation vs energy consumption -the lower the better-.

Let's start by modeling the memory bandwidth usage when one IP accesses data from one HBM bank and let's assume that the number of replicated IPs is given as N_{IP}. The *aocl* compiler reports the frequency, $F_{N_{\text{IP}}}(j=1)$ at which this implementation is synthesized, where j represents the number of IPs per memory bank (1 in this case). Being W the width (Bytes) of one HBM bank, then we can estimate the ideal bandwidth per IP and per bank as,

$$BW_{ideal}(j=1) = F_{N_{\text{IP}}}(j=1) \cdot W \tag{4.23}$$

Using the FPGA Dynamic Profiler for OpenCL tool [161] we obtain $BW_m(j=1)$, the measured memory bandwidth per IP and per HBM bank in our implementation. Now, we can define the memory bandwidth usage per IP and per HBM module, σ, as

$$\sigma = \frac{BW_m(j=1)}{BW_{ideal}(j=1)} \tag{4.24}$$

Through exhaustive experimentation with different implementations in which we keep N_{IP} fixed but increase the number of IPs that can access one HBM module ($j > 1$) while decreasing the number of active HMB modules, i ($i = N_{\text{IP}}/j$), we find that factor σ represents a good estimation of the memory bus occupancy while the bus is not saturated, because the measured memory bandwidth per HBM bank increases linearly with the number of IPs per bank. Thus, we model or estimate the memory bandwidth per HBM as,

$$\begin{aligned} BW(j \geq 1) &= j \cdot BW_m(j=1) \\ &= j \cdot \sigma \cdot BW_{ideal}(j=1) \end{aligned} \tag{4.25}$$

In case of bus saturation, there is not headroom for one additional IP accessing data from a HBM module, in other words, the maximum achievable bandwidth per bank is,

$$BW_{max} = (1-\sigma) \cdot BW_{ideal}(j=1) \tag{4.26}$$

In summary, from Eqs. 4.25 and 4.26 we compute the aggregated memory bandwidth for i HMB modules as,

$$\begin{aligned} ABW(i, j \geq 1) &= j \cdot \sigma \cdot BW_{ideal}(j=1) \cdot i \\ &\leq (1-\sigma) \cdot BW_{ideal}(j=1) \cdot i \end{aligned} \tag{4.27}$$

Figure 4.5 represents a case of study when $N_{\text{IP}} = 24$, where the IPs are distributed among different number of memory banks (i): from 24 HBM banks (i.e. $j = 1$) to 2 HBM banks ($j = 12$). The experimental setup is detailed in Section 4.4.1.

In Figure 4.5(a), lines depict the aggregated memory bandwidth (MB/s) for different number of HBM modules. The dashed line is the aggregated memory bandwidth from Equation 4.27, while the solid line is the actual aggregated memory bandwidth measured for each configuration. The values for $i = 24$ represent in fact $BW_{ideal}(j = 1)$ (Equation 4.23) and the measured $BW_{meas}(j = 1)$, which our model uses to compute σ (Equation 4.24). As we see, the model predicts accurately the behavior of the HBM system, being the deviation below 11%. The figure also shows the application throughput (elements/ms), and that the aggregated memory bandwidth is a good proxy of the performance behavior.

Figure 4.5(b) depicts measured power (Watts) -lines- and energy (Joules) -bars-. The main power components (due to the IPs and the UIB[3] bus consumption) demonstrate that decreasing the number of HBM banks reduce power consumption. Thus, it is advisable to deploy the minimum number of memory banks that guarantees optimal memory bandwidth usage. This is our definition of optimal number of memory banks. As shown in Figure 4.5(a), maximum memory bandwidth is sustained from 24 to 6 banks. In fact, the energy consumption is the minimum for the same range of memory banks. Reducing the number of HBM modules below 6 increases the number of IPs per memory bank, which causes the saturation of each module memory bandwidth. As a consequence, both the aggregated memory bandwidth (thus, throughput) and energy consumption degrade.

With this model we compute the optimal number of memory banks. From Eqs. 4.25 and 4.26 we firstly find the optimal number of IPs per bank, j_{opt}, that is the maximum number of IPs that can access data from one HBM module without saturating the bus (ensuring this way optimal memory bandwidth usage),

$$j_{opt} = max(j, 1) \quad : \quad j \leq \left\lfloor \frac{1 - \sigma}{\sigma} \right\rfloor \tag{4.28}$$

Once we have found the optimal number of IPs per HBM bank, we calculate the optimal number of memory banks that ensure the maximum aggregated memory bandwidth as,

[3]Universal Interface Bus that powers the HBM DRAM.

$$i_{opt} = \left\lceil \frac{N_{\text{IP}}}{i_{opt}} \right\rceil \tag{4.29}$$

In our case of study we find that $\sigma = 0,167$, so $j_{opt} = 4$ and $i_{opt} = 6$. From Figure 4.5(a) we corroborate this finding.

Please note that in the experimental section we have follow this methodology for selecting the optimal number of active HBM banks and optimal number of IPs per bank for any given number of IPs.

4.4. Experimental results

4.4.1. Experimental setup

The experimental evaluation has been conducted on a CPU+FPGA platform.

The processor architecture and software details of each platform can be found in Tables 4.1 and 4.2, respectively.

Table 4.1: Platform details.

Microarchitecture	**CPU Intel Core**
Processor	i7-7820X
Number of cores	8
Clock Speed	3.6 GHz
Max Turbo Frequency	4.50 GHz
Main memory	128GB DDR4
Cache L3	11 MB
Litography	14 nm
Max TDP	140 W
Microarchitecture	**FPGA Intel Stratix 10 MX**
Main memory	16GB HBM
HBM Memory Banks	32 banks (512MB per bank)

All results (performance, energy, and profiling metrics) report the median value of 5 runs. The performance metric is throughput (elements per millisecond) and energy is reported in Joules. The normalized standard deviation for throughput (energy) measurements is always below 3% (4%). Unless otherwise stated,

Table 4.2: Software details (CPU & FPGA).

Operating System	CentOS 7.2.1511
Intel FPGA SDK for OpenCL	version 19.3
Intel OpenCL	1.0
Intel TBB and VTune	2020 Update 3
Processor Counter Monitor	201902
GCC Compiler	4.8.5.

our heterogeneous runs simultaneously exploit 8 CPU cores and, as motivated in section 4.4.2.1, 40 IP/FPGA compute units. Energy results were obtained using the Processor Counter Monitor (PCM) library for the CPU part, and a the self-developed Stratix-Monitor library [166] for the FPGA device.

We consider 4 time series of different sizes: 2^{17} (131072), 2^{18} (262144), 2^{19} (524188) and 2^{20} (1048576). These time series are random-walk time series that are commonly used for benchmarking in time series analysis algorithms [167].

The *Fastfit* scheduler discussed in section 4.2.2 is invoked with $\rho = 0.99$ and $\delta = 0.2\%$, and it is compared with three previous schedulers that were initially devised for CPU+GPU platforms [146]:

- *Static*: it splits the iteration space in two chunks at once: one for the CPU cores and the other for the accelerator. The size of these two chunks is user-defined and provided via the `offload_ratio` input argument. If `offload_ratio`=0 (0%) the CPU process the whole iteration space, and so does the FPGA if it is equal to 1 (100%). The CPU chunk is divided in equally sized sub-chunks for each CPU core, i.e. $chunkCore = chunkCPU/NumCores$.
- *Dynamic*: it lazily splits the iteration space dynamically. Each time the FPGA is idle, it takes a chunk from the iteration space. The size of this chunk is user-provided using the `chunkFPGA` input argument. The CPU cores also take chunks of the iteration space, but now $chunkCore = chunkFPGA/\varphi$ where φ is the relative speed of the FPGA w.r.t. the CPU core (i.e. if the FPGA is 2x faster than a CPU core, $\varphi = 2$, so `chunkCore` is half the size of `chunkFPGA`). A guided self-scheduling [168] is used when there are not enough remaining iterations to enforce the previous equations.
- *Logfit*: it also dynamically splits the iteration space, but the user does not provide a constant `chunkFPGA` size. On the contrary, this `chunkFPGA` size

is now an adaptive variable that is automatically computed by the scheduler following a logarithmic fitting strategy that has been proved beneficial for irregular codes on CPU+GPU systems [146].

4.4.2. FPGA-only evaluation

In this subsection, we evaluate the performance of our schedulers when the FPGA is the only device computing the matrix profile. Heterogeneous executions are considered in the next subsection.

4.4.2.1. Kernel replication exploration

One of the FPGA optimizations described in 4.1 was kernel replication [161]. This optimization results in a better utilization of the FPGA resources, leading to performance gains if there is enough bandwidth to feed all the replicated kernels or IPs. We explore the performance for different number of replicated IPs applying the methodology given in section 4.3 for selecting the optimal number of active HBM banks and optimal number of IPs per bank for each case. The results for time series input 2^{20} are shown in Figure 4.6. The maximum number of IPs that fit in our FPGA is 42, but as can be seen in Figure 4.6 maximum performance is obtained for 40 IPs. Smaller time series exhibited the same behavior. Unless explicitly stated, from now on, we fix the number of FPGA IPs to $N_{\text{IP}} = 40$.

4.4.2.2. Fixed-point arithmetic approach

During the development of our FPGA implementation we also considered the possibility of using fixed-point arithmetic instead of floating-point, since fixed-point could report better performance and/or energy consumption. We have manually implemented fixed-point arithmetic using bit shifting. The float variables are converted to integers by bit shifting, then we operate on them and convert them back to float at the end of the calculation. We tried with different number of bits for the decimal part of the number, in order to optimize accuracy. Figure 4.7 shows the performance of fixed-point arithmetic compare to floating-point arithmetic.

As the fixed-point implementation do not use the floating-point hardware units available in the FPGA, more FPGA logic blocks are necessary to implement the fixed-point version which results in a maximum number of IP replications of 14 IPs. As detailed in Section 4.4.2.1, the floating-point implementation can

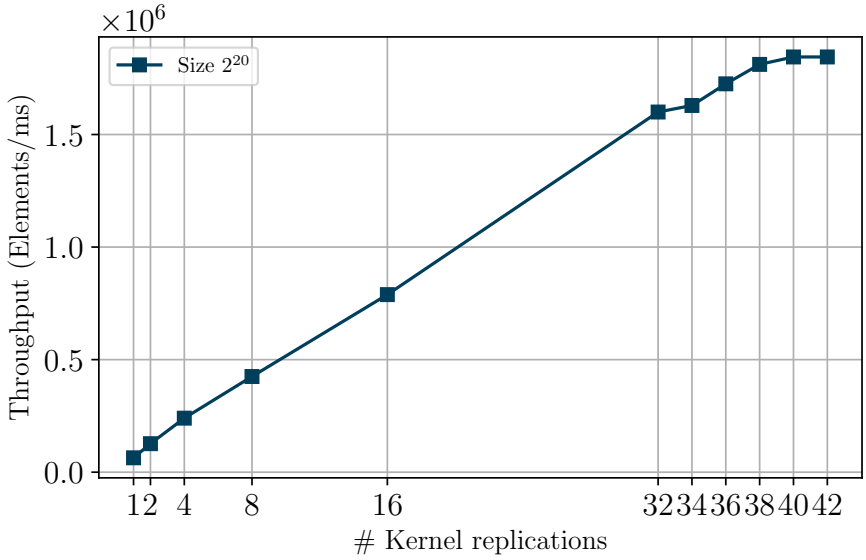


Figure 4.6: Exploring the number of kernel replications, N_{IP}, for the 2^{20} time-series and only-FPGA execution.

accomodate up to 42 IPs. Fixed-point arithmetic improves floating-point up to 164.11% when comparing the same # of IPs, 14, but it achieves only 12.95% of improvement when comparing their maximum performance (14 IPs in fixed-point vs 40 IPs in floating-point). However, the problem of fixed-point arithmetic is the accuracy loss. Even with all the combinations of bits for decimal and integer part we tried, we could not achieve 100% of accuracy in results. This is a restriction of our design that we can not avoid, as we can not miss any motif or discord in the matrix profile computation. In FPGA, designing fixed-point arithmetic can be achieved if it is manually implemented using *HDL* (Hardware Description Level). However, our work focus on performance productivity from the programmer point of view, applying only *HLS* (High Level Synthesis), so we have not considered a manual implementation of fixed-point arithmetic. Hence, with this accuracy loss, even though the is an small improvement in performance when using fixed-point, we kept the floating-point implementation as the only one valid for our FPGA implementation in the rest of the work.

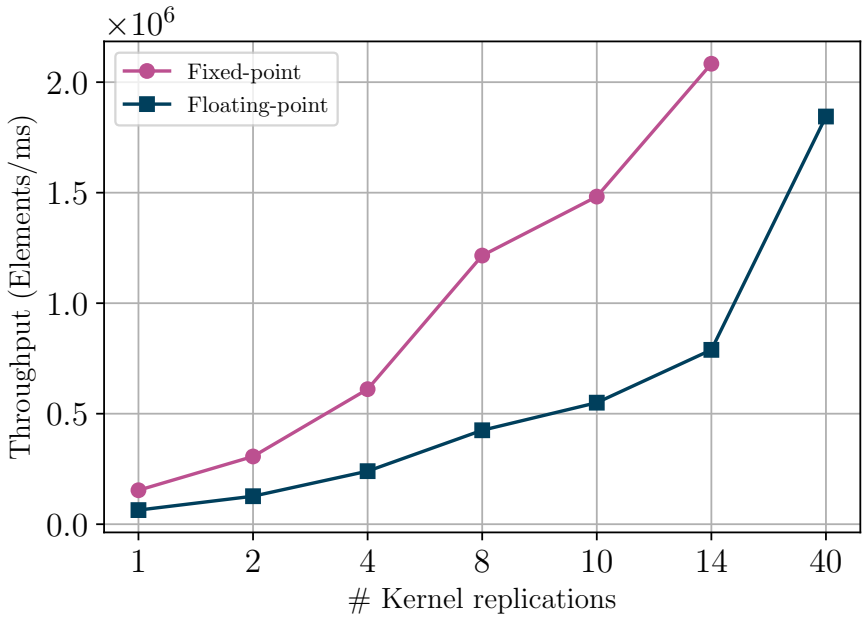


Figure 4.7: Exploring the performance of Fixed-point arithmetic versus Floating-point arithmetic for a 2^{20} time-series, only-FPGA execution and varying the number of kernel replications, N_{IP}.

4.4.2.3. Evaluation of partition strategies at device-level

As explained in section 4.2.3, once the system-level scheduler assigns an FPGA chunk to the FPGA, the device-level scheduler has to partition this chunk among the different IPs. In this section, we quantitatively validate the throughput improvements due to the use of our proposed *Balanced* partition strategy with respect to a naive *Block* one that is not aware of the different diagonal lengths.

Figure 4.8 shows the impact in the throughput for the different system-level schedulers: *Dynamic*, *Logfit* and *Fastfit*. Although in this experiment only the FPGA is used (there are no CPU cores collaborating in the computation), we assess these three system-level schedulers since they produce different FPGA chunk sizes as well as a significant number of chunks. In the figure, we plot the throughput of the FPGA for the *Dynamic* scheduler when configured with FPGA chunk sizes from 2^6 to 2^{20} (being 2^{20} the whole iteration space in our largest time series). Since *Logfit* produces variable FPGA chunk sizes during the

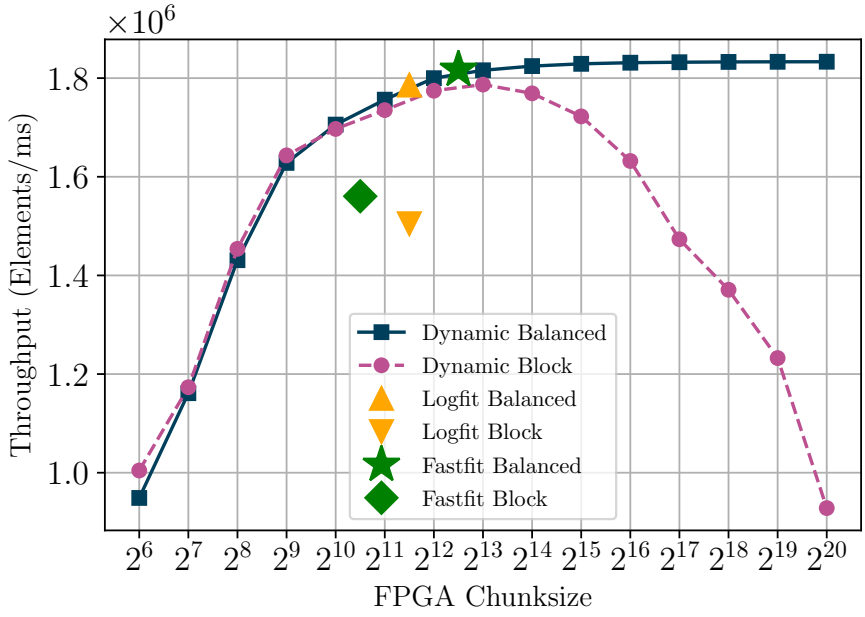


Figure 4.8: Device-level scheduler exploration for different schedulers and a time series size of 2^{20}. X-axis represents the FPGA chunk size and the Y-axis the throughput. The higher the better.

computation, the average chunk size is shown in Figure 4.8, while for the *Fastfit* scheduler we depict the FPGA chunk size found in the *Training Phase*. Note that since *Logfit* and *Fastfit* schedulers compute the FPGA chunk size depending on the FPGA throughput, using *Balanced* or *Block* partition strategy in the device-level scheduler, may have an impact on the FPGA chunk sizes found, as well as on performance.

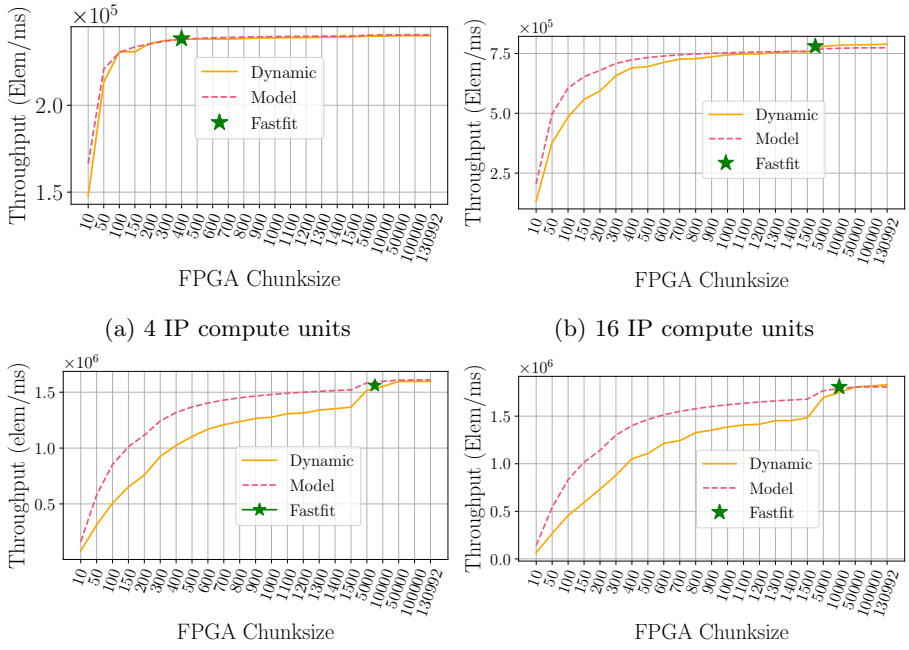
Figure 4.8 evidences that the workload unbalance of the *Block* partition can have a remarkable impact on the throughput, especially for large FPGA chunk sizes, as we see in the *Dynamic* scheduler for chunk sizes larger than 2^{14}. On the other hand, when using the *Block* strategy, *Logfit* and *Fastfit* tend to find smaller FPGA chunk sizes than using *Balanced*. This is because the FPGA throughput measured during the training (in both schedulers) is smaller due to load unbalance among the IPs which results in sub-optimal chunk size estimation.

In the three schedulers, using the *Balanced* partition strategy always achieves the best performance. In summary, *Balanced* results in 97.37% better throughput

than *Block* for the highest chunk size in the *Dynamic* scheduler, and 18.66% and 16.45% improvements in throughput for *Logfit* and *Fastfit*, respectively. This result motivates us to keep using the *Balanced* partition strategy in the rest of the evaluation.

4.4.2.4. Validation of *Fastfit* model

In this subsection, we validate the *Fastfit* model described in section 4.2.2. This model is devised to predict the FPGA throughput for any FPGA chunk size. For it, after obtaining F, IL and DL, we use equation 4.18 to compute the throughput for any chunk size CF. Figure 4.9 shows the estimated throughput (`Model`) computed for $\rho = 0.99$ vs. the actual measured one (`Dynamic`) for different FPGA chunk sizes and different number of replicated IPs for the 2^{17} time series.



(a) 4 IP compute units

(b) 16 IP compute units

(c) 32 IP compute units

(d) 40 IP compute units

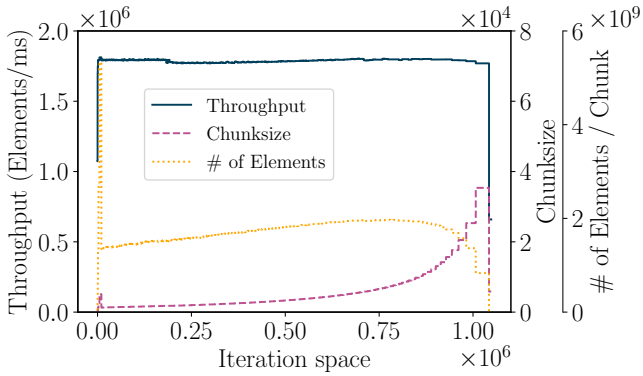
Figure 4.9: Comparison of *Fastfit* model simulation, *Dynamic* and *Fastfit* executions for time series size 2^{17}. X-axis represents the FPGA chunk size and the Y-axis the throughput. The higher the better.

In addition, in the figure we mark with a green star the throughput for the estimated near-optimal FGPA chunk size, CF_ρ, obtained after the Training Phase in a real execution of *Fastfit*. It is worth remarking that: i) the model is accurate, especially for smaller number of IPs where the device-level scheduler overhead and potential load unbalance among IPs have a less noticeable impact on the real throughput; ii) the execution of *Fastfit* end up using an FPGA chunk size that results in an almost optimal throughput. For instance, the throughput predicted by the model for the near-optimal chunk size is between 97%-99% of the actual measured throughput for the selected chunk size. Similar accuracy was achieved for different input sizes; iii) the FPGA chunk sizes found leave room for CPU collaboration and CPU+FPGA heterogeneous co-execution; and iv) all in all, our model allows to obtain the desired throughput out of the FPGA without having to perform the manual exploration required by *Dynamic*.

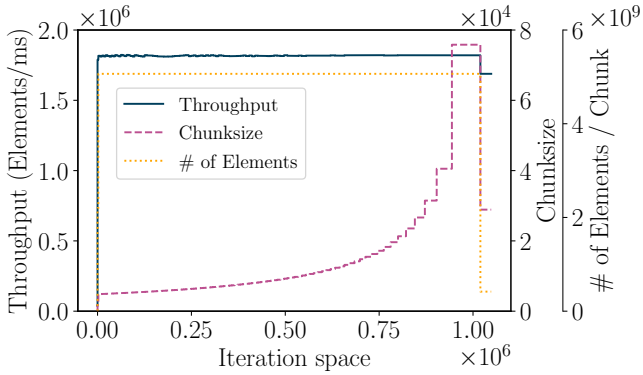
Although it was initially devised for CPU+GPU platforms and irregular algorithms, *Logfit* [146] is a related scheduler that can also save the exploration time that was needed with *Dynamic*. Both *Logfit* and *Fastfit* avoid the manual exploration of a suitable chunk size using a two phases scheme: a *Training phase* and a *Exploitation phase*. However *Logfit* spends more time than *Fastfit* in the *Training Phase* and continuously recompute new FPGA chunk sizes during the *Exploitation phase*, which can suppose additional overhead. In particular, the *Training phase* of *Logfit* requires more time because it samples the throughput obtained with monotonically increasing chunk sizes until the throughput stop growing. Using four of the previous samples, it computes the logarithmic function that fit these samples and with this it can select the near-optimal accelerator chunk size (see [146] for more details). The pipeline architecture of FPGA IPs allows the *Fastfit* simple model to be accurate enough with only two throughput samples and no logarithmic fitting.

In order to better understand the different behavior between *Logfit* and *Fastfit*, Figure 4.10 shows the evolution of the throughput (`Throughput`) and FPGA chunk size using two metrics: number of diagonals per chunk (`Chunksize`=CF) and number of total elements in the chunk (accumulating all the elements of all the diagonals in the chunk, `# of Elements/Chunk`=N_F). In Figure 4.10(a) we can see a glitch, in the `Chunksize` and `# of Elements/Chunk` curves, at the beginning of the iteration space were several samples are needed until we can move on to the *Exploitation phase*. In Figure 4.10(b), although it is hardly noticeable, we only test two different chunks sizes that let us to quickly move to the next phase.

The *Exploitation phase* of *Logfit* is also more complex and introduces more overhead since it keeps adapting the accelerator chunk size to suits to through-



(a) Logfit



(b) Fastfit

Figure 4.10: *Logfit* and *Fastfit* evolution of the throughput, FPGA chunk size and total number of elements of the chunks for the 2^{20} time series. X-axis represents the iteration space.

put changes that can be frequent in irregular codes. However, the *Exploitation phase* of *Fastfit* assumes the code is regular and that the chunk size estimated in the previous phase is valid for the whole iteration space. As we can see in Figure 4.10(b), *Fastfit* sustains a more stable throughput and an almost constant `# of Elements/Chunk` that is directly proportional to the workload per chunk of iterations and higher than the workload per chunk assigned by *Logfit*. This explains the higher average throughput observed in *FasFit*. Note that in both schedulers, the chunk size increases to keep the `# of Elements` constant since the diagonals are becoming shorter as we sweep the iteration space. Also note

that the throughput can take a performance hit at the end of the iteration space when there are not enough iterations to fully utilize the FPGA pipeline.

4.4.3. Evaluation of heterogeneous CPU+FPGA executions

In this section, we validate the four system-level heterogeneous schedulers, *Static*, *Dynamic*, *Logfit* and *Fastfit* when using both the CPU (8 cores) and the FPGA co-executing simultaneously. We first focus on the obtained performance and later on the energy efficiency.

4.4.3.1. Heterogeneous scalability

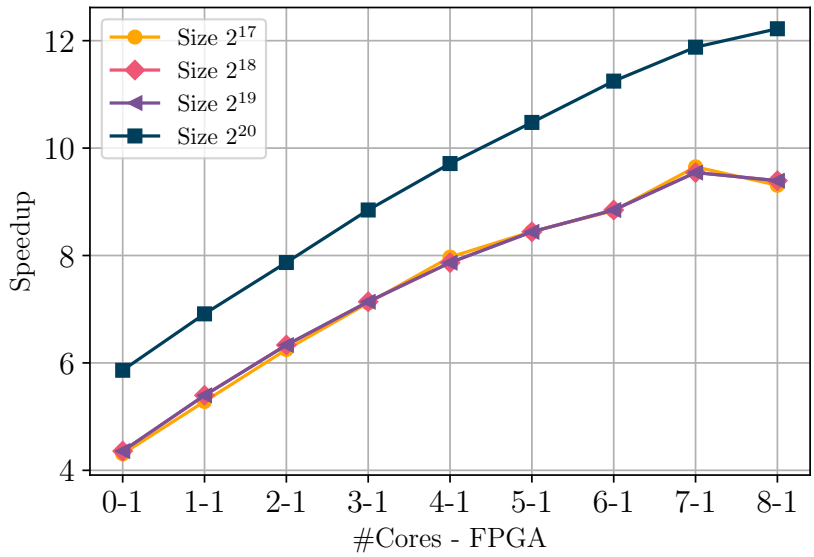


Figure 4.11: Heterogeneous Performance Scalability using *Fastfit*. Speedup for different Input Size respect to 1 Core-CPU. The higher the better.

Before analyzing the heterogeneous schedulers implementations, it is interesting to show the scalability of the heterogeneous execution. Figure 4.11 shows the speedup obtained for 4 time series sizes and 9 configurations: 0-1 that is the FPGA-only implementation (0 CPU cores and 1 FPGA), and $x-1$ being x the number of cores that goes from 1 to 8. The FPGA alone yields up to 6x better

performance (for the larger time series), but the speedup keep increasing when adding CPU cores to help in the computation. The scalability with the number of cores is not linear (around 6x for 8 cores) due to parallel overheads, memory and cache sharing issues (SCAMP is a memory bound application) and also due to the CPU frequency reductions that are imposed when more cores are working.

4.4.3.2. Performance analysis

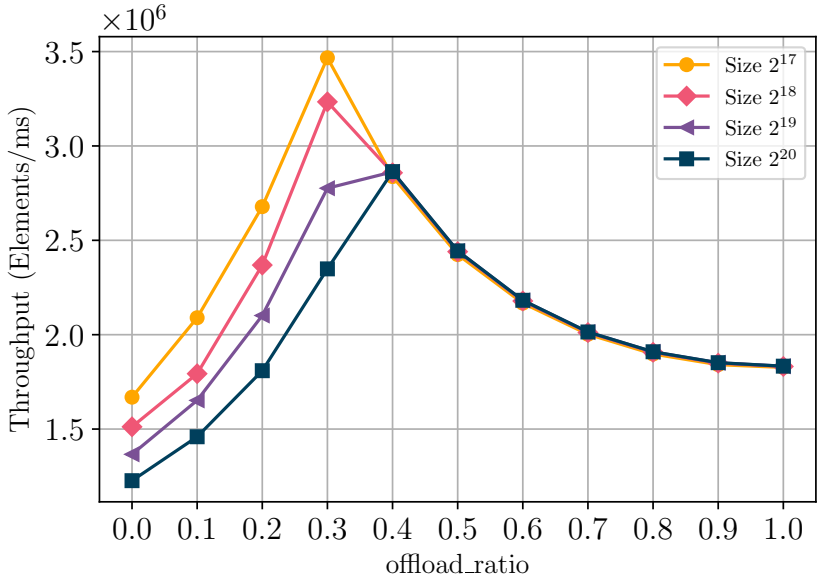


Figure 4.12: Throughput for *Static* and different time series. X-axis represents the percentage of iterations (as a ratio) offloaded to the FPGA. The higher the better.

Remember that *Static* requires that the user provides an `offload_ratio` stating the percentage of iterations offloaded to the FPGA. Figure 4.12 shows the throughput obtained for `offload_ratio` between 0 (only CPU execution) and 1 (only FPGA execution) and four time series with different sizes. After this manual exploration, we found that the smaller time series exhibit maximum throughput when 30% of the iteration space is computed on the FPGA, but for larger time series it is better to offload 40% of the iteration space to the FPGA and compute the rest on the CPU. Different time series may require different `offload_ratio` values and a more precise search might pay off (in steps of 1%

instead of 10% as in the figure) although more time should be devoted to the exploration.

As we saw in Figure 4.8, *Dynamic* also requires an offline profiling in order to find a suitable FPGA chunk size, that for the 2^{20} time series ends up being 2^{13} diagonals. Note that larger chunk sizes result in similar throughput but makes less likely to achieve CPU+FPGA work-sharing and load balance. *Logfit* and *Fastfit* automatically find, without user intervention, the suitable sizes for the FPGA and the CPU cores, at a small training overhead.

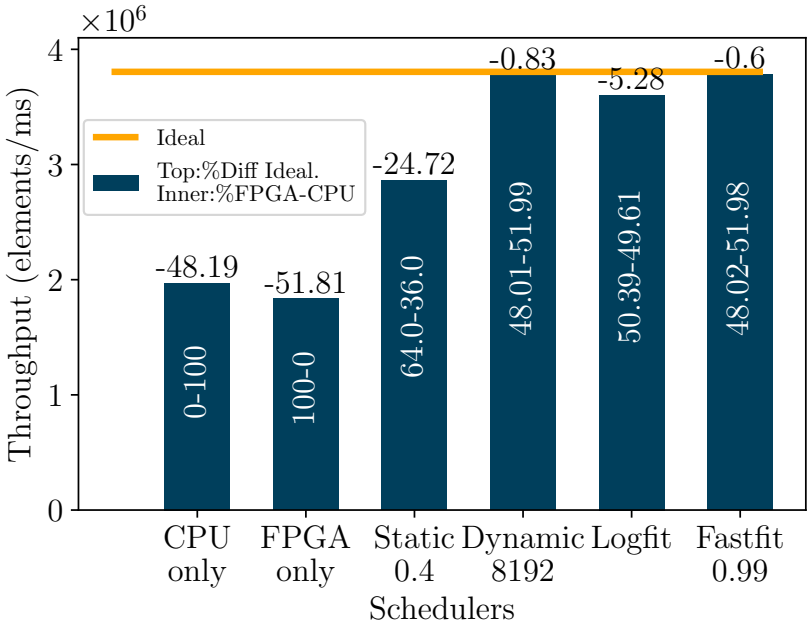


Figure 4.13: Throughput comparison for all schedulers and 2^{20} input using the FPGA and 8 CPU cores. The numbers inside the bars indicate the percentage of Elements (not diagonals) computed on the FPGA (bottom) and CPU (top). The numbers above the bars are the percentage of performance degradation with respect to the ideal throughput (CPU only + FPGA only throughput) represented as an horizontal line.

In order to compare the performance of all the evaluated schedulers, Figure 4.13 shows a throughput comparison for the best results obtained with each scheduler and the 2^{20} time series. The number inside the bars indicates the percentage of Elements processed on each device (FPGA-CPU), so the CPU-only and

FPGA-only executions show 0-100 and 100-0, respectively. Note that although in *Static* we set `offload_ratio`=0.4 which distributes 40% of the iterations (diagonals) to the FPGA and 60% to the CPU, the FPGA ends up processing 64% of the Distance Matrix Elements since the first 40% of the diagonals are larger than the remaining 60%. The upper orange horizontal line indicates the Ideal throughput, estimated as the aggregation of the CPU-only throughput and the FPGA-only one. This ideal throughput does not account for the unbalance and scheduling overheads so it is an upper bound used to quantitatively estimate the impact of these overheads.

The best *Dynamic* execution is obtained for FPGA chunksize equal to 8192, `chunkFPGA`=2^{13} as we saw in Figure 4.8. With this manual configuration it only looses 0.83% of performance w.r.t. the ideal due to the partitioning overhead, although it requires the offline exploration to find the best `chunkFPGA` input argument. *Logfit* departs 5.28% of the ideal due to the scheduler overhead (training and logarithmic re-fitting) that were mentioned in section 4.4.2.4. However *Fastfit* delivers almost ideal performance, automatically finding a very good initial FPGA chunksize of 9,455 diagonals. Subsequent FPGA chunk sizes are updated just to maintain a constant workload (number of Elements) as explained in section 4.2.2.1. This small difference with respect to *Dynamic* makes *Fastfit* delivers a slightly better throughput, highlighting an almost negligible overhead (0.6%). Similar results have been obtained for different input sizes. This finding, along with the advantage of avoiding the manual search of a near-optimal FPGA chunk size, turn *Fastfit* into an excellent scheduler for co-execution of regular algorithms on CPU+FPGA platforms.

4.4.3.3. Energy analysis

Figure 4.14 depicts a breakdown of energy efficiency (in Elements per Joule) and energy consumption (in Joules) in the same conditions explained in the previous section. FPGA energy is measured thanks to an in-house library (publicly available [166]) built on top of the BMC (Board Management Controller) library provided by the FPGA vendor (BittWare). Energy efficiency has been computed dividing the number of computed elements by the total number of Joules consumed.

First, paying attention to the one-device only results at the left of Figure 4.14, it can be observed that the FPGA exhibits the highest energy efficiency and the lowest energy consumption. The CPU requires almost 3x more energy to carry out the same computation. Now, the energy consumed by the heterogeneous schedulers is roughly proportional to the workload processed by each device (CPU

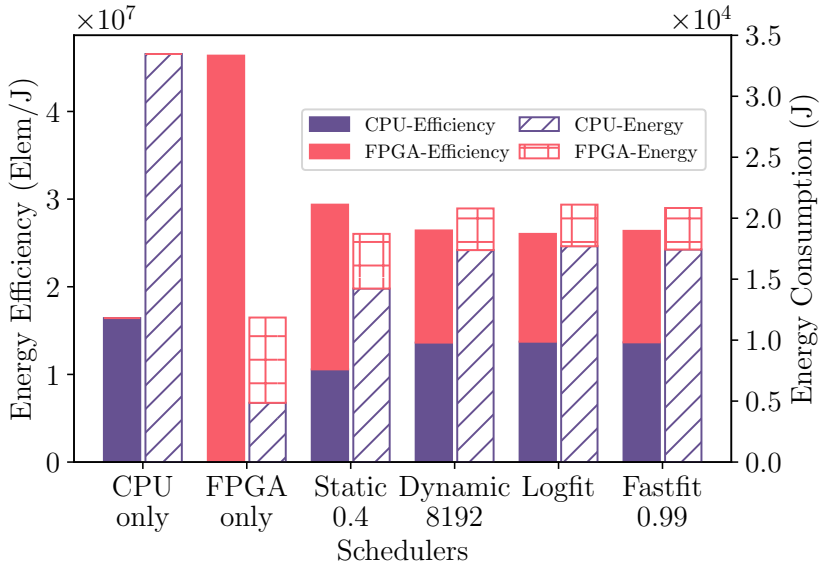


Figure 4.14: Energy metrics for all schedulers and 2^{20} time series using the FPGA and 8 CPU cores. Solid bars represent the Energy efficiency (the higher the better), whereas patterned bars depict Energy consumption (the lower the better).

and FPGA) as pointed out in Figure 4.13. For example, *Static* offload more work to the FPGA (64% of the elements) and consequently exhibits better energy efficiency than *Dynamic*, *Logfit* and *Fastfit*. Actually, energy consumption and energy efficiency in these last three schedulers are similar. Due to *Logfit* being the slowest of the last three schedulers, it also consumes more energy than *Dynamic* and *Fastfit*.

Table 1 includes the relevant data already presented in previous charts. Summarizing, *Fastfit* is the best scheduler if our goal is to achieve maximum performance. Although *Dynamic* also achieves good results, let's recall that in this case the user needs to explore offline exhaustively all possible chunk sizes to find the near optimal, whereas in *Fastfit* the best chunk size is automatically discovered at runtime. On the other hand, if the target is energy consumption, it is better to switch off the computation on the CPU cores and resort to the FPGA-only execution. Again, similar conclusions can be obtained for different input sizes.

Table 4.3: Summary of performance, energy consumption and energy efficiency. Diff. represents the degradation with respect ideal or best. The optimal implementation for each criterion in boldface.

	CPU 8 cores	**FPGA Only**	**Static**	**Dynamic**	**Logfit**	**Fastfit**
Throughput (Elements/ms)	$1.97 \cdot 10^6$	$1.83 \cdot 10^6$	$2.86 \cdot 10^6$	$3.7 \cdot 10^6$	$3.6 \cdot 10^6$	$\mathbf{3.78 \cdot 10^6}$
%Diff. Ideal Throughput (CPU+GPU)	-48.19%	-51.81%	-24.72%	-0.83%	-5.28%	**-0.6%**
Energy consumption (Joules)	$3.34 \cdot 10^4$	$\mathbf{1.18 \cdot 10^4}$	$1.87 \cdot 10^4$	$2.07 \cdot 10^4$	$2.11 \cdot 10^4$	$2.08 \cdot 10^4$
%Diff Best Energy consumption	182.32%	**0.0%**	57.88%	75.47%	78.11%	75.78%
Energy efficiency (Elements/J)	$1.64 \cdot 10^7$	$\mathbf{4.67 \cdot 10^7}$	$2.93 \cdot 10^7$	$2.64 \cdot 10^7$	$2.6 \cdot 10^7$	$2.67 \cdot 10^7$
%Diff Best Energy efficiency	-64.58%	**0.0%**	-36.66%	-43.01%	-43.86%	-43.11%

4.4.4. OpenCL vs oneAPI implementation

In this section we present the evaluation of the productivity from a programmer point of view between OpenCL and oneAPI rather than the runtime performance. By the time this work was developed, there was no BSP (Board Support Package) compatible with oneAPI for our FPGA. Hence, this work have been developed using OpenCL as a programming model for our heterogeneous implementation. As detailed in Section 2.3.2, oneAPI is an open, cross-architecture programming model developed by Intel with the objective to unify the code development for different heterogeneous architectures. The main goal is to be able to write the same code and compile it for different architectures, making life easier for programmers of heterogeneous platforms [169].

The metrics used to compare the programmer productivity using OpenCL and oneAPI are the Programming Effort and Cyclomatic complexity. The Programming Effort (PE) is a function of: (1) the number of unique operands and unique operators; (2) the total number of operands and operators. The operators

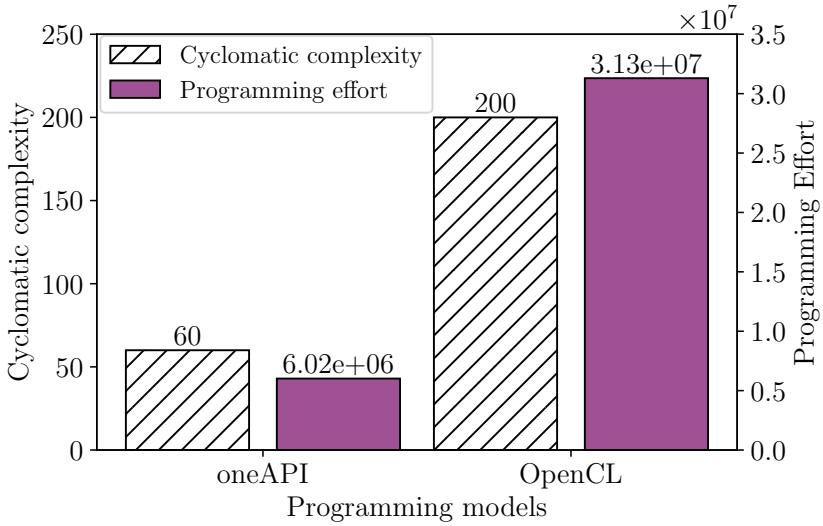


Figure 4.15: Comparison between oneAPI and OpenCL in terms of Cyclomatic Complexity and Programming Effort for the code of this project. Solid bars represent Programming Effort metric, while pattern bar represent Cyclomatic complexity metric. The lower the better.

are the symbols that affect to the value of the operands, while the operands are the identifiers and constants. The Cyclomatic complexity (CC) is the number of predicates plus one. Higher values for PE and CC indicate that it is more complicated for a programmer to code such program. The research by [170] tackle this same analysis for a Value Iteration Problem on a CPU + GPU heterogeneous platform, achieving interesting results with a 5.1x difference in PE but similar CC for the oneAPI and OpenCL implementations.

Figure 4.15 present the results of our comparison between oneAPI and OpenCL using PE and CC. Our analysis shows that in terms of PE, OpenCL presents 5.3x more PE than oneAPI, while in terms of CC this difference is reduced to 3.3x. Using both metrics we conclude that the oneAPI implementation is much easier to implement. As a future work we will propose to also evaluate the performance of the oneAPI implementation once a oneAPI compatible BSP is released for our FPGA.

4.5. Conclusions

In this chapter, we study the problem of efficiently executing a state-of-the-art time series algorithm class -The Matrix Profile- on a heterogeneous platform comprised of CPU + High Performance FPGA with integrated HBM (High Bandwidth Memory). The geometry of the algorithm (a triangular matrix walk) and the FPGA capabilities pose two challenges. First, several replicated IPs can be instantiated in the FPGA fabric, so load balance is an issue not only at system-level (CPU+FPGA), but also at device-level (FPGA IPs). Second, the data that each one of these IPs accesses must be carefully placed among the HBM banks in order to efficiently exploit the memory bandwidth offered by the banks while optimizing power consumption.

To tackle the first challenge we propose a novel hierarchical scheduler named *Fastfit*, to efficiently balance the workload in the heterogeneous system while ensuring near-optimal throughput. Our scheduler consists of a two level scheduling engine:

1. The system-level scheduler, which leverages an analytical model of the FPGA pipeline IPs, is used to find the near-optimal FPGA chunk size that guarantees optimal FPGA throughput and from that the CPU chunk size that ensures load balance among devices.
2. A geometry-aware device-level scheduler, which is responsible for the effective partitioning of the FPGA chunk into sub-chunks assigned to each FPGA IP. To deal with the second challenge we propose a methodology based on a model of the HBM bandwidth usage that allows us to set the minimum number of active banks that ensure the maximum aggregated memory bandwidth for a given number of IPs.

Through exhaustive evaluation, we validate the accuracy of our models, the efficiency of our intra-device partition strategies and the performance and energy efficiency of our *Fastfit* heterogeneous scheduler, finding that it outperforms state-of-the-art previous schedulers by achieving up to 99.4% of ideal performance.

Summarizing, the main contributions of this chapter are:

- We present an efficient heterogeneous CPU + FPGA implementation that reduces the execution time, contributing with, to the best of our knowledge, the first FPGA implementation of a matrix profile algorithm using High Level Synthesis, HLS, that reduces energy consumption with respect to the only-CPU approach.

- We develop a methodology based on a analytical model to optimize the memory bandwidth usage of HBM Banks. Our model finds the minimum number of HBM banks to ensure maximum bandwidth while reducing power consumption.
- We propose a hierarchical scheduler, named *Fastfit*, that: (1) at inter-device level an analytical model calculates a near optimal partitioning of the work for the FPGA and the CPU cores, that efficiently balances the workload among devices; and (2) at intra-device level, it uses a custom static partition of the diagonals of the matrix profile to balance workload for all FPGA IPs, so that they finish their assigned partition at the same time.

5 Skyline computation on Heterogeneous CPU + GPU processors

In the previous two chapters, we propose scheduling strategies to efficiently optimize time series computation through the matrix profile algorithm, using different heterogeneous architectures: CPU + GPU and CPU + FPGA. However, in this chapter, we tackle a different massive data application: The skyline problem. The skyline, initially introduced in [86], is an optimization operator widely used for multi-criteria decision making. It allows to minimize a n-dimensional dataset into the smallest subset, usually using as a reduction metric the *minimum* value for each dimension. In order to increase the skyline performance, it is key to avoid the all-to-all comparison between points. To that end, two approaches are usually adopted: (1) sorting-based or (2) partitioning-based. The main disadvantage of sorting-based algorithms is that for high dimensional skylines, the methodology generates a large candidate buffer, causing performance degradation due to brute-force quadratic search. State-of-the-art sequential algorithms use recursive, point-based partitioning approaches. The current state-of-the-art multi-core algorithm, *Hybrid* [23], is a point-based method that dynamically constructs a quad-tree with the skyline points. One optimization of this algorithm is that it flattens the tree into an array structure for better access patterns, and also it processes points in blocks (tiles) to improve parallelism. However, point-based strategies are not well suited to heterogeneous architectures. For instance, in [23] the tree is constructed on the fly, incrementally, and sequentially, so frequent synchronization points are necessary to accommodate the sequential insert phase, a strategy that adversely affects performance on the GPU. More-

over, the uncontrolled branching in the tree traversal tends to serialize execution within each warp on account of branch divergence. On the other hand, the current state-of-the-art algorithm for GPU architectures, *SkyAlign* [24], is based on a statically-defined quad-tree, being the key algorithmic idea that points are physically sorted by grid cells and statically partitioned, and threads are mapped onto that sorted layout. Besides, the actual computation is loosely ordered with d carefully placed synchronization points (being d the number of dimensions). This type of order simultaneously achieves good spatial locality, homogeneity within warps, and independence among threads, in particular when the number of dimensions (d) is high. This strategy creates more predictable tree traversals that minimizes branch divergence. As a novelty in our work, as we are targeting an integrated GPU, we have designed a new implementation of the *SkyAlign* algorithm based on SYCL [43]. SYCL allows us to use the same algorithm and source code both on the CPU and the GPU (contrary to CUDA), obtaining reasonable performance on both devices. SYCL advocates for the single source code approach, which enables to target multiple devices using the same programming model in order to have a cleaner, portable, and more easy to maintain applications. In our implementations, we leverage oneAPI [45] that is a promising framework that simplifies programming heterogeneous architectures by providing several libraries and a unified programming language DPC++ (Data Parallel C++) [46] able to target CPUs, GPUs and FPGAs. DPC++ incorporates the SYCL language and extensions for Unified Shared Memory (USM), ordered queues, reductions and subgroups, which we exploit to take advantage of architectural features on our heterogeneous platform.

The mentioned skyline algorithms are designed to compute skylines over static datasets rather than dynamic ones that occur in data streaming environments. Dynamic data streaming represents a continuous stream of received data points. In the append-only data streams, sets of data points are removed when they expire. For instance, such type of streams are those of wireless sensors networks, where the data collected prior a specific time interval are discarded because they are not representative in comparison with the new readings of sensors. In that context, current sequential or parallel approaches [110, 108, 109, 171] compute the skyline on sliding windows for the n most recent points that arrive, producing a stream of skyline updates with the arrival of new points. In the context of analytic applications that process multi-source data streaming queries, or applications in data exploration and multi-criteria decision making that project the multi-dimensional data into different subset of the attributes (i.e., some subspaces of interest) [95, 172], we find that the data stream is provided with diverse independent queries for each of which the computation of the skyline operator is

required. Examples include the development of smart technologies in the context of the rapid advancement in the Internet of Things (IoT) ecosystem.

In this chapter we tackle the problem of computing the skyline operator over a stream of independent data queries using as target architecture a heterogeneous system comprised of a multi-core CPU and an integrated GPU, which to the best of our knowledge it has not yet been addressed. In our work, we propose a heterogeneous graph-based engine, called SkyFlow based on oneAPI [45], which is capable of efficiently schedule the data queries computations among the devices while ensuring near-optimal throughput. Our proposal adapts to different streaming scenarios using two heterogeneous approaches: Coarse-grained (SkyFlow-CG) and Fine-grained (SkyFlow-FG).

SkyFlow-CG computes concurrently one query per device. In this work, we experimentally validate the performance of our SYCL implementations, both on the GPU and the CPU finding that although the SYCL code is portable, it is not "performance portable" because it performs better on the GPU than on the CPU. In fact, as we will see in Section 5.1.5, for our platform the original OpenMP-based *Hybrid* implementation is faster than the SYCL-based *SkyAlign* on the CPU. Thus, SkyFlow-CG adopts a hybrid strategy: each device runs the algorithm best suited to the specific features of the corresponding device, it is, *Hybrid* on the CPU (implemented with OpenMP) and *SkyAlign* on the GPU (implemented with SYCL). During our research, we found that specific datasets perform better under *Hybrid* on the CPU than under *SkyAlign* on the GPU, or vice versa, depending on the distribution of points in the dataset and its spatial structure, size, or number of dimensions. Thus, as we aim at optimizing system performance and resource utilization in the context of a stream of independent data queries, we must devise a scheduling strategy that at runtime is able to consider the arriving data query characteristics and the occupancy of the resources to dispatch the skyline computation to the appropriate device. In this chapter we propose different scheduling strategies and evaluate and discuss their performance and optimality for different streaming scenarios.

On the other hand, SkyFlow-FG represents a heterogeneous CPU+GPU solution for the skyline computation in which each single dataset query is split between the CPU and the GPU devices. For it, we start from the SYCL-based *SkyAlign* implementation that runs both on the CPU and GPU. The main challenge now is to find the optimal dataset partition for each arriving data query at runtime. We also evaluate different partitioning strategies for different streaming scenarios.

In our experimental evaluations we find that our heterogeneous approaches always outperform baselines implementations that only use one device. In fact, they outperform only-GPU and only-CPU baselines up to 5.19x and 6.86x, respectively. These results tell us that exploiting both devices with our heterogeneous solutions is usually more profitable than using just one device. We will also discuss under which streaming scenarios is advantageous to use SkyFlow-CG, and in which ones SkyFlow-FG delivers better results.

The rest of the chapter is organized as follows. Section 5.1 introduces the required background and the skyline algorithms. Section 5.2 presents our heterogeneous approaches for computing the skyline over a stream of data queries. Section 5.3 describes the schedulers and partitioning strategies devised to optimize the heterogeneous solutions. Section 5.4 discusses the experimental results, ending with conclusions in Section 5.5.

5.1. Theoretical Background

5.1.1. Definitions

Let's compute the skyline corresponding to the dataset (or data query) S of n points, p_i, with d dimensions. The value of p_i in a given dimension δ is known as $p_i[\delta]$.

Definition 1: Dominance.

A point p dominates another q, $p \prec q$, if the following condition is satisfied: $\forall i \in [0, d-1] : p[i] \leq q[i]$ and $\exists j \in [0, d-1] : p[j] < q[j]$; that is, for every dimension, p is less than or equal to q and there exists at least one dimension in which p is strictly less than q. This operation is called dominance test (DT).

Definition 2: Skyline. A skyline can thus be defined as the subset of points in a dataset S that are not dominated by any other point in the dataset, i.e., $SKY(S) = \{p \in S \mid \nexists q \in S : q \prec p\}$.

Definition 3: Incomparability. Two points $p, q \in S$, are incomparable, $p \sim q$, if $p \not\prec q$ and $q \not\prec p$, that is, if p and q do not dominate each other.

The smaller the number of DTs, the better the work-efficiency and the faster the skyline computation. In the worst case, for a dataset of size n, the algorithm will have quadratic cost with $n \cdot (n-1)/2$ DTs. Avoiding DTs resulting in incomparability reduces the computational cost of the problem. Sorting and partitioning based algorithms work towards this goal. In sorting-based algo-

rithms, traversing the sorted points reduces the number of DT operations. In partitioning-based, a partitioning of the space avoids DT between points that are known to be incomparable because their corresponding partitions are also incomparable.

Figure 5.1 shows a partitioning-based example with a small dataset comprising only 4 points out of which A, B and D belong to the skyline set. As we can see, a DT between pairs of points A and C and A and D is unnecessary since A is in region $M = 01$ that dominates in X-axis, but C and D are in region $M = 10$ that dominates in Y-axis, resulting in the incomparability of all the points in both regions. On the other hand, a DT between points C and D is required because both are in the same region M=10 and, hence, comparable. The point C is dominated in both dimensions by the point D and deleted as a skyline point. Finally, a DT between points A and B results in a incomparability since point A dominates in X-axis while point B dominates in Y-axis. The same result comes when comparing points B and D. The median mask M and quartile mask Q will be covered in detail in Sect. 5.1.3.

Point	M	Q
A	01	10
B	11	00
C	10	11
D	10	10

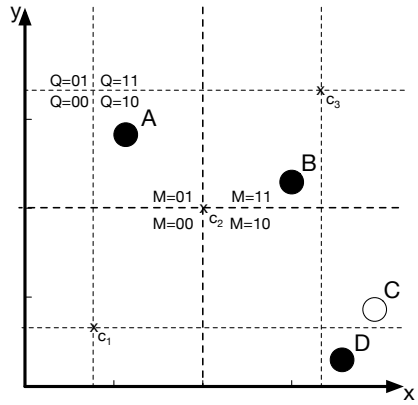


Figure 5.1: Dataset and skyline example. Skyline points are A, B and D. Pivot points, c_i, and mask values, median mask M and quartile mask Q, represent the static space partitioning created. The X-axis (Y-axis) is the first (second) dimension.

As introduced in Sect. 2, this work is based on two skyline algorithms: (1) *Hybrid* [23] initially implemented in OpenMP [173] is the state-of-the-art for multi-core CPU architectures. We will refer to it as *OpenMP-CPU*. And (2) *SkyAlign* [24] initially developed in CUDA [174] is the state-of-the-art for GPU architectures, having considerable potential for scalability and heterogeneous im-

plementation. We have ported the implementation to SYCL, because the integrated GPU does not support CUDA, so from now on, we will refer to it as *SYCL-GPU*. Depending on the dataset, the CPU algorithm can be faster than the GPU one, or the other way around. Next, we briefly introduce both algorithms and compare their performance.

5.1.2. OpenMP-CPU algorithm

Algorithm 6 sketches the CPU algorithm, which combines two techniques to save DTs: sorting and partitioning.

Algorithm 6: OpenMP-CPU algorithm

```
   Input: S=Dataset of n points p and d dimensions.
   Output: Skyline of S: SKY(S)
 1 SKY(S) ← ∅
 2 Prefilter, partition and sort S
 3 while S ≠ ∅ do
 4 |   Q ← next α points of S
 5 |   S ← S\Q
 6 |   foreach i ∈ [0, Q.size) (in parallel) do
 7 |   |   if ∃p ∈ SKY(S) : p ≺ Q[i] then
 8 |   |   |   Mark Q[i] as dominated
 9 |   Remove dominated Q[i] from Q
10 |   foreach i ∈ [0, Q.size) (in parallel) do
11 |   |   if ∃j ∈ [0, i) : Q[j] ≺ Q[i] then
12 |   |   |   Mark Q[i] as dominated
13 |   Remove dominated Q[i] from Q
14 |   Append Q to SKY(S)
15 return SKY(S)
```

The algorithm is divided into three blocks. The first one is represented by the line 2 that carries out a prefiltering, partitioning and sorting steps that we describe next. (1) **Prefiltering** performs a fast parallel comparison of points in chunks according to their Manhattan norm (L1), easily pruning dominated points; (2) **Partitioning** does the partition of the multi-dimensional space based on a pivot point, p_v. The p_v is the point with the median value of L1, which must necessarily be a skyline point. A mask, m, is assigned to each point for each

dimension such that, $m[i] = (p[i] < p_v[i]?0 : 1)$[1]. p_v divides the dataset into 2^d regions so that the binary mask of each point identifies its corresponding region. Avoiding comparisons between points in incomparable regions reduces the DTs needed. (3) **Sorting** is carried out according to the binary mask and the L1 norm. This sorting maintains the property that $p \not\prec q$ if p precedes q in the sort order. See more details in [23].

After this preprocessing, the resulting dataset, S, is traversed in blocks of α points (line 4). A study carried out in Sect.VII-C of [23] fixed the optimal size of alpha in 1024 (see [23] for more details). Each block, Q, is processed by two consecutive parallel loops. A first parallel stage (lines 6-8) compares each point p of the block with the points of the global skyline known so far, to check if any of them dominates p. After the parallel stage, a synchronization stage sequentially eliminates dominated points from the iteration space (line 9). The second parallel stage (lines 10-12), compares the surviving points of the previous stage among them. The second synchronization stage sequentially eliminates dominated points in line 13. The points that pass through this two sieves are added to the global skyline, $SKY(S)$, in line 14. Thanks to the partitioning stage, binary mask comparisons are used to reduce the number of the more expensive DT operations. The sorting stage ensures that: 1) once a point is appended to the $SKY(S)$ it will not leave this set since no subsequent point in S will dominate it; and 2) points that are likely pruning others are processed earlier (which remove the DT operations corresponding to these early pruned points).

This process is repeated for all the blocks in S updating the global skyline after processing each block. Although not explicitly indicated in Algorithm 6, vectorization is used for the implementation of the DT operations (lines 7 and 11).

5.1.3. SYCL-GPU algorithm

The *SYCL-GPU* follows a different approach in order to avoid the synchronization stages that keep a global skyline in the *OpenMP-CPU* algorithm. On the GPU these synchronization steps have a higher impact on performance. As in the *OpenMP-CPU* alternative, similar optimizations are also considered: static space partitioning and dataset sorting.

[1] C language ternary operator: $m[i]$=0 if condition holds, and 1 otherwise.

5.1.3.1. Space partitioning

The partitioning divides each dimension of the dataset in quartiles and median. This is, three global pivot points (instead of just one as in *OpenMP-CPU*) are defined for each dimension: first quartile, c_1, median, c_2, and second quartile, c_3 (see Figure 5.1). All GPU threads share the information of these three defined global pivot points. It should be noted that these points are probably virtual (i.e. c_1, c_2, and c_3 may not belong to the dataset as it happens with the pivot point in the *OpenMP-CPU* algorithm). This partitioning results in binary masks to classify the dataset points in two nested levels. At the first level, a binary mask M is assigned to each point, corresponding to its position relative to the median (equivalent to the pivot point in the *OpenMP-CPU*). On a second level, another binary mask Q is assigned, corresponding to its position relative to the first or third quartile (whichever is relevant for each point). The process is repeated for each dimension. These two binary masks per point ease locating each point's partition.

Figure 5.1 shows an example with a dataset of 4 points partitioned according to the quartiles and median points. Formally, we define the assignment of binary mask values to points using space partitioning. Let us denote $c_{i\delta}$ the i quartile of the δ dimension, with $c_{2\delta}$ being the median. If we start from an example point p_i, we can have the following mask configurations, depending on where the point is located with respect to the median and the quartiles:

$$M_i[\delta] = 0, Q_i[\delta] = 0 \Leftrightarrow p_i[\delta] < c_{1\delta}$$

$$M_i[\delta] = 0, Q_i[\delta] = 1 \Leftrightarrow p_i[\delta] \in [c_{1\delta}, c_{2\delta})$$

$$M_i[\delta] = 1, Q_i[\delta] = 0 \Leftrightarrow p_i[\delta] \in [c_{2\delta}, c_{3\delta})$$

$$M_i[\delta] = 1, Q_i[\delta] = 1 \Leftrightarrow p_i[\delta] \geq c_{3\delta}$$

For example, point B has median mask $M_B = 11$ because in its x dimension it is above the median value, c_{2x}, and in the y dimension is also above the median, c_{2y}. Its quartile mask is $Q_B = 00$ because in the x dimension is below the c_{3x} quartile, and below the c_{3y} quartile as well for the y dimension.

Once the partitioning and binary masks have been defined, the next step is to establish the equations on which the MT will be based. The purpose of these equations is to establish non-dominance relations, which avoid having to perform DT operations.

There are two levels of application of these equations: 3 equations where only the median is required and if these fail, two additional equations where the quartile masks are consulted.

Let us assume points p_i and p_j, with binary median masks M_i and M_j. We have the order information, $|M_j|$ and $|M_i|$, that is, the number of bits of the masks with value 1. The equations using the first level of information (just the median), are:

$$(M_j|M_i) > M_i \Rightarrow p_j \not\prec p_i \tag{5.1}$$

$$|M_i| < |M_j| \Rightarrow p_j \not\prec p_i \tag{5.2}$$

$$|M_i| = |M_j|, M_i \neq M_j \Rightarrow p_j \sim p_i \tag{5.3}$$

The equation 5.1 tests whether M_j has any bit set to 1 that is not in M_i. If that is the case then there exists some dimension in which $p_i[\delta] < p_j[\delta]$ and hence $p_j \not\prec p_i$. In Figure 5.1 it can be seen how this equation is satisfied for points A and B.

The equation 5.2 covers the particular case of the equation 5.1 where M_j has more bits set to 1 than M_i. In this case the equation 5.1 is also satisfied and the same conclusion is reached. Again this equation can be checked in Figure 5.1 for points A and B.

Likewise, the equation 5.3 is another special case, where the order of both masks is the same. Under this assumption, and in the case that the masks are not equal (that they do not have the same bits set to 1), it necessarily causes that both points are incomparable. This equation is applied to points A and C or points A and D in Figure 5.1.

For cases where the median-level binary masks M_j and M_i do not determine a non-dominance relationship ($p_j \not\prec p_i$), the quartile-level binary masks (Q_i and Q_j) are applied with these two equations:

$$\begin{aligned} M_j \preceq M_i, (((M_j| \sim M_i)\&Q_j)|Q_i) > Q_i \\ \Rightarrow p_j \not\prec p_i \end{aligned} \tag{5.4}$$

$$M_j = M_i, (Q_j|Q_i) > Q_i \Rightarrow p_j \not\prec p_i \tag{5.5}$$

The equation 5.4 approximates the equation 5.1, starting from the prior knowledge of the median masks $M_j \preceq M_i$, which states that $M_j[\delta] \leq M_i[\delta] \ \ \forall\delta$. The expression $M_j| \sim M_i$ gives results for dimensions where $M_j = M_i$, so it indicates in which dimensions the points p_j and p_i are on the same side of the median. These would be the dimensions for which the above median-level equations would not give a result in this case (in the others it would be satisfied that $p_j < p_i$). $(M_j|M_i)\&Q_j$ selects the bits that cannot be solved by the median mask equations and in which p_j is greater than the quartile. If p_i is less than the quartile for any of these cases, then $p_j \not\prec p_i$. In Figure 5.1 this relationship can be seen for points B and C to determine that $p_C \not\prec p_B$. At first, p_C is better than p_B for the second dimension and equal for the first, with respect to the median of each dimension. Hence, the first assumption is satisfied, $M_C \preceq M_B$. If we look at the equation, we have that $(M_C| \sim M_B)\&Q_C = 10$, and that $((M_C| \sim M_B)\&Q_C)|Q_B = 10$, which is greater than $Q_B = 00$. These result also can be demonstrated visually as we see that p_C is above p_B in the quartile. The equation is satisfied and therefore $p_C \not\prec p_B$.

The equation 5.5 is a particular case of the equations 5.4, where the median masks are equal and therefore $(M_j|M_i)\&Q_j = Q_j$. This can be seen in Figure 5.1 for points C and D, with identical median mask ($M_C = M_D = 10$). Applying the equation we get $Q_C|Q_D = 11|10 = 11$, and, hence, $11 > Q_D$. Therefore, $p_C \not\prec p_D$.

5.1.3.2. Algorithm description

Once the points have been classified according to their region in the space (identified by both masks M and Q) we can leverage the fact that points located in incomparable regions are also incomparable and avoid the corresponding DT. In essence, we trade DTs for MTs (mask tests). Note that MTs are computationally cheaper than DTs since they only perform one binary operation between two integers, while a DT requires $2{\cdot}d$ operations (see Definition 1). The sorting step has the same advantages that we mentioned in the previous section.

Algorithm 7 presents a description of the GPU algorithm. For a more precise explanation we refer the reader to [24]. The algorithm is divided in two main stages: (1) preprocessing; and (2) the main loop.

Preprocessing: The preprocessing includes prefiltering, partitioning and sorting. Prefiltering is carried out in lines 1-10. The idea is to first find a threshold, τ, that is calculated as the minimum of the maximum value in all the dimensions of each point. For example, in the Table of Figure 2.12, the

Algorithm 7: SYCL-GPU algorithm

Input: S=Dataset of n points p and d dimensions.
Output: Skyline of S $SKY(S)$

1 $\tau \leftarrow min_{p \in S}(max_{i \in [0,d)}(p[i]))$
2 $S \leftarrow \{p \in S | \exists i \in [0,d) : p[i] \leq \tau\}$
3 **foreach** *dimension* $\delta \in [0,d)$ **do**
4 | Sort S by dimension δ
5 | $c_{i\delta} \leftarrow S[\lfloor i * |S|/4 \rfloor][\delta], i \in 1,2,3$
6 **foreach** *point* $p_i \in S$ *(in parallel)* **do**
7 | **foreach** *dimension* $\delta \in [0,d)$ **do**
8 | | $M_i[\delta] \leftarrow (p_i[\delta] > c_{2\delta})$
9 | | $Q_i[\delta] \leftarrow (p_i[\delta] > (M_i[\delta] ?\ c_{3\delta} : c_{1\delta}))$
10 Sort S according to M and $|M|$
11 **foreach** *levels* $l \in [0,d)]$ **do**
12 | **foreach** *point* $p_i \in S$ *(in parallel)* **do**
13 | | **if** $|M_i| > l$ **then**
14 | | | **foreach** $M : |M| = l \wedge (M|M_i) = M_i$ **do**
15 | | | | **foreach** $p_j \in S,\ M_j = M$ **do**
16 | | | | | **if** $(((M_j| \sim M_i) \& Q_j)|Q_i) > Q_i$ **then**
17 | | | | | | **if** $p_j \prec p_i$ **then**
18 | | | | | | | Mark p_i dominated; terminate thread
19 | | **else**
20 | | | **foreach** *point* $p_j \in S,\ M_j = M_i$ **do**
21 | | | | **if** $(Q_j|Q_i) = Q_i$ **then**
22 | | | | | **if** $p_j \prec p_i$ **then**
23 | | | | | | Mark p_i dominated; terminate thread
24 | Remove dominated points from S
25 | SKY(S) $\leftarrow$ SKY(S) $\cup \{p_i \in S : |M_i| = l\}$
26 **return** $SKY(S)$

maximum values are computed by row, resulting in the vector {3, 3, 2}, from with the minimum is $\tau = 2$. In parallel, each point is compared with the threshold and if it has no value less than the threshold, that point is dominated and eliminated. For example, in Figure 2.12, point A does not have any value smaller than 2, so it is pruned. Once this prefiltering of points finishes, the static partitioning is created (lines 3-9). In lines 3-5 the medians and quartiles are calculated, assigning the binary masks (median and quartile) to each point in lines 6-9. Finally the

dataset is sorted according to M and $|M|$ in line 10 so that the data layout matches the control flow described next.

Main loop: The algorithm's main loop goes from lines 11 to 25. A detailed explanation of the main loop can be found in [24] but for what matters in this chapter, suffices it to say that the outer sequential loop with iterator l, level, has d iterations (line 11). In each iteration, a parallel loop compares every point, p_i, with the rest of the points, and those with median mask of order[2] l and non dominated (not pruned by MT equations 5.1-5.5 or DT) are progressively[3] added to the skyline solution. First, the median masks, M, are used (lines 13-15 and line 20). If the median mask does not help, then the quartile masks, Q, are compared (line 16 and 21). Only if both MT fail, the corresponding DT is carried out (line 17 and 22). If the point is dominated (line 18 and 23), it is marked as dominated, and the thread terminated. At the end of each iteration of the outer loop, the dominated points are removed. The non-dominated points for that order are added to the skyline.

Note that the update of $SKY(S)$ (line 25) requires a synchronization at each outer iteration, but contrary to the *OpenMP-CPU*, the number of synchronization points is equal to the number of dimensions, which is smaller than the number of synchronization points in *OpenMP-CPU* for large datasets.

5.1.4. SYCL-GPU migration from CUDA to oneAPI

The GPU algorithm was initially developed in CUDA [24]. The code has been migrated from CUDA to oneAPI. Since oneAPI is a multi-device language, the current implementation of the algorithm allows its execution on any Intel device (CPU, GPU or FPGA). We took advantage of this feature of oneAPI to implement SkyFlow for both CPUs and GPUs, and to combine both implementations in a heterogeneous GPU + CPU version as we will see later.

To facilitate the migration of the original CUDA code the Intel DPC++ Compatibility Tool (DPCT) [175] has been used. Intel has developed this new tool to automate the conversion of CUDA code to oneAPI. DPCT allows conversion from individual source codes to entire projects via command line, with a reasonably high automatic conversion coverage (over 80% of the CUDA source code was automatically converted to oneAPI in our experience). DPCT also generates

[2]Being the order of the median masks, $|M_i|$, the number of 1's in the mask M_i (i.e. $|M_i| == l$)

[3]Progressive skyline algorithms yield solution points on the go, in contrast to algorithms that provide the solution at once after completion.

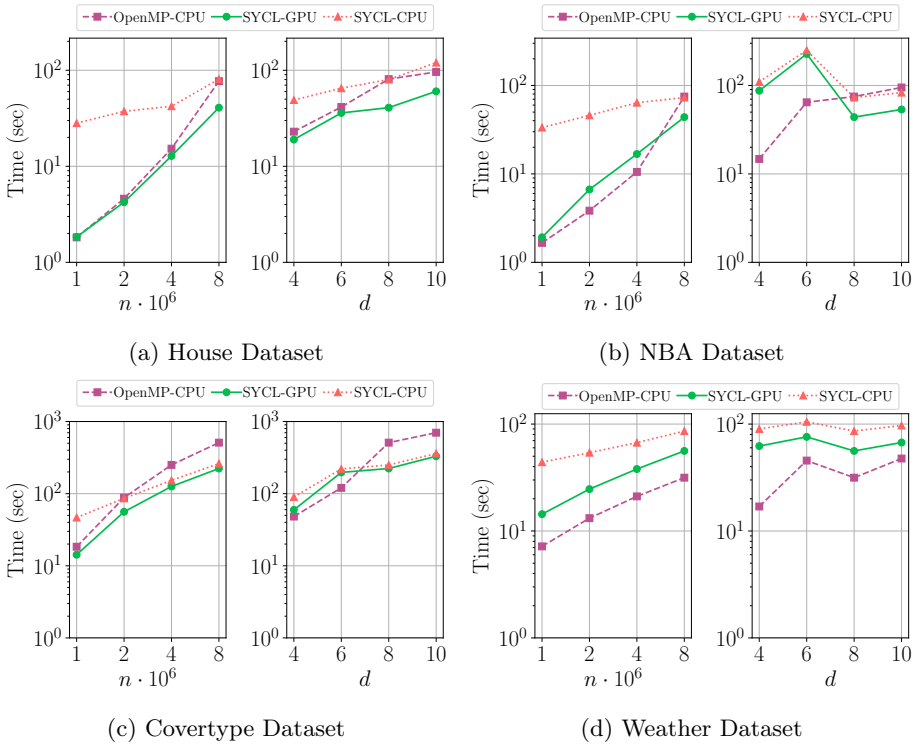
comments in the generated source code (and in the terminal) to facilitate debugging and conversion of the parts of the code that the tool could not translate automatically. In our case, the tool could generate almost all the structure of the oneAPI code. However, the more complex parts of the code (such as the main loop) required manual intervention. The reason is that the original CUDA version uses CUDA libraries and tools with no direct equivalent in oneAPI. Specifically, the original code uses the BOOST library for thread handling and the THRUST library for processing specific functions (such as sort and parallel_reduce). In our implementation, all BOOST functions have been successfully ported using the Standard Template Library (STL) and C++17. Likewise, GPU-parallelized functions with THRUST have been migrated to existing equivalents in the oneAPI Data Parallel Library (DPL).

5.1.5. Initial performance assessment

Up to now we have seen two different implementations that solve the same problem: computing the skyline of a dataset (or data query). *OpenMP-CPU* is optimized for the CPU, whereas *SYCL-GPU* comes from a GPU optimized CUDA code. However, now that we have the GPU version written in SYCL, we can take advantage of the portability exhibited by SYCL implementations. In SYCL the computation is enqueued to a device, which is configured using a `device_selector` object. Just by changing the `device_selector` from GPU to CPU and re-compiling, the SYCL code can run on the CPU. This means, that we actually have three versions: *OpenMP-CPU*, *SYCL-GPU* and *SYCL-CPU*, being the last two the same implementation but targeting different devices.

Remember that our overarching goal is to accelerate the skyline computation on a heterogeneous CPU+GPU architecture. This requires first to assess the performance of the three versions. To this end, we have executed the three implementations on a octa-core Intel i9-9900K CPU that includes an integrated GPU (more details of the test-bed in Section 5.4).

Figure 5.2 shows the execution times for *OpenMP-CPU*, *SYCL-GPU* and *SYCL-CPU* algorithms and four datasets (described in Section 5.4). For each dataset, the subfigure on the left fixes $d = 8$ and changes n from $1 \cdot 10^6$ to $8 \cdot 10^6$. The right subfigure fixes $n = 8 \cdot 10^6$ and changes d from 4 to 10. From the plots we can conclude two main takeaway messages. First, that there is no device (CPU or GPU) that always dominates the other. Depending on several factors (points distribution in the dataset, n and d) the CPU or the GPU can be the fastest device. Second, out of the two CPU implementations (OpenMP-



(a) House Dataset (b) NBA Dataset

(c) Covertype Dataset (d) Weather Dataset

Figure 5.2: Execution time for the three implementations (*OpenMP-CPU*, *SYCL-GPU*, *SYCL-CPU*) and four datasets (the lower the better).

CPU and SYCL-CPU), the OpenMP-CPU is usually faster (except for House and Covertype datasets and larger values of n and/or d). Although the SYCL implementation is portable, it is not "performance portable" and considering that it was developed with the GPU in mind (it derives from a CUDA implementation) it is not optimized for the CPU architecture in this platform. Therefore, from now on, we will use the *OpenMP-CPU* version on the CPU and the *SYCL-GPU* version on the GPU, unless contrary stated.

A work-efficiency study conducted in [24] reports that, for higher dimension datasets, *SkyAlign* requires less DTs than *Hybrid*. In these cases, *Hybrid* carries out more DT operations (less work-efficiency) for the sake of exploiting more parallelism. Thus, the *SkyAlign* algorithm (and *SYCL-GPU*) strives to offer both more parallelism and work-efficiency for higher dimension datasets. Therefore, except for Weather dataset, we corroborate that *SYCL-GPU* outperforms

OpenMP-CPU for high dimensionality queries. A deeper study of the behaviour of *SYCL-GPU* for the Weather dataset shows that: (1) the preprocessing stage is not able to prefilter any point before entering the main loop; and (2) in the main loop the number of MTs is significantly smaller than in the other datasets and, hence, higher the number of DTs. This degrades the work-efficiency of *SYCL-GPU*, causing higher execution times in comparison to *OpenMP-CPU*.

On the contrary, the *OpenMP-CPU* algorithm is less affected by the spatial distribution of the points. Its two parallel loops exhibit more regularity than the *SYCL-GPU* algorithm's main loop. The fact that *OpenMP-CPU* processes the dataset in blocks results in a better use of the cache hierarchy. Besides, the CPU is less affected by data and control divergence. On the other hand, the prefiltering step is less aggressive than in the *SYCL-GPU* one. To sum up, the skyline computation is highly irregular, heavily depending on the dataset configuration (distribution of points in the space, size, number of dimensions) and on the particular algorithm and target architecture.

5.2. SkyFlow: Heterogeneous Skyline over a stream of data queries

Now that we have an efficient implementation of the skyline algorithm for the CPU (*OpenMP-CPU*) and the GPU (*SYCL-GPU*), our goal is to devise an optimal graph-based engine to deal with a stream of data queries on a CPU+GPU architecture, like the Intel i9-9900K described in Section 5.4. To this end, we rely on the FlowGraph classes provided by the Threading Building Block library [47] (part of the oneAPI [45] framework), that is cleverly designed to ease the optimization of data flow problems.

In the following sections we describe several data flow solutions, including the only-CPU and only-GPU baseline as well as two CPU+GPU heterogeneous approaches.

5.2.1. Baseline SkyFlow

As a baseline, we have developed a "single-device" version of our data flow approach, SkyFlow-CPU and SkyFlow-GPU, that only exploit the CPU or the GPU, respectively. Figure 5.3 shows the FlowGraph that we describe next.

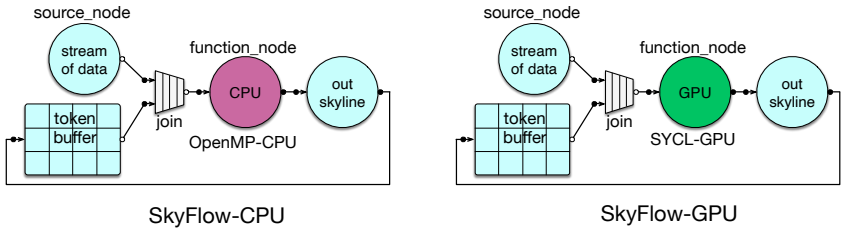


Figure 5.3: Structure of the baseline SkyFlow graphs.

As we see in the figure, the graph is composed of a number of nodes connected by edges that we classify in three main sections: (1) *Source*, (2) *Execution*, and (3) *Output*.

- *Source*: It comprises a `source_node`, a `token buffer` and a `join_node`. The *source_ node* generates the stream of data queries from which we require the skyline. However the stream is not fed directly into the graph to avoid oversubscribing the HW if the input data rate is much higher than the processing data rate. The idea is to rely on a token-based approach to limit the resource consumption. To that end, a buffer is pre-filled with a number of tokens and a `join_node` will forward a dataset (or data query) down the graph only if it can be paired with a token. That way, the maximum number of data queries in flight is limited to the number of tokens. At the output node, the token is recycled back into the buffer so that a new dataset can be injected into the *Execution* section of the graph.

- *Execution*: In this baseline solution, it contains a single `function_node` that computes the skyline. The SkyFlow-CPU configures this node to run the *OpenMP-CPU* algorithm whereas the SkyFlow-GPU runs the *SYCL-GPU* code.

- *Output*: Finally, the last node takes care of saving the resulting skyline, optionally checks that the computation is correct and recycles the token to enable the processing of a new dataset. It is also possible to use this node to merge several partial skylines in the case a dataset is partitioned into several blocks and processed separately by SkyFlow.

On our current platform with a multi-core CPU and an integrated GPU we have validated that over-subscription is avoided by having just two tokens: one

computing a skyline, and a second in the queue waiting to be dispatched as soon as the device becomes available. In SkyFlow-CPU the *OpenMP-CPU* parallel algorithm takes care of fully utilizing all the CPU cores.

5.2.2. Coarse-Grained Heterogeneous SkyFlow

For our first heterogeneous approach we follow the easiest strategy that consists in combining in the *Execution* section the GPU and CPU algorithms concurrently, as depicted in Figure 5.4.

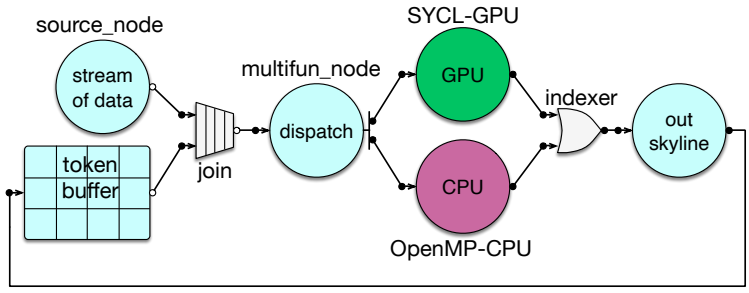


Figure 5.4: Structure of Coarse-Grained SkyFlow graph, SkyFlow-CG.

We call this version Coarse-Grained SkyFlow, SkyFlow-CG, because it computes a whole dataset (or data query) either on the GPU or on the CPU. In the next section we describe a Fine-Grained approach in which a single dataset is split between the CPU and the GPU. Note that the *Execution* section now begins by a `dispatch` node that, for each dataset, decides whether it should be processed by the *SYCL-GPU* or the *OpenMP-CPU* algorithms. The number of tokens should be incremented in order to allow for at least two data queries in flight. Contrary to the `join` node, the `indexer` node (that comes after the GPU and CPU nodes) asynchronously passes incoming tokens to the next node without waiting for an input at each entry port, that way enabling that CPU tokens and GPU tokens traverse the flow graph at its own pace. This means that the resulting skylines can be computed out-of-order, but this is not a problem if we tag each result with the ID of the corresponding data query. If the output order is relevant it is always possible to insert before the output node a `sequencer` node [47] that would reorder the skyline results.

Assuming that the application bottleneck is in the *Execution* section (the one we are optimizing by exploiting both the CPU and the GPU at the same time), we implement two queues in the `dispatch` node (one per device). Now the problem is to feed these two queues so that the total execution time is minimized (throughput maximized). Two scheduling strategies have been considered and will be covered next: Work Conserving scheduling (WC) and Heterogeneous Earliest Finish Time (HEFT).

The goal of a Work Conserving scheduling is to keep all the scheduled devices busy. To that end, it strives to keep the queues of the devices with the same length (number of pending tasks). In our implementation we maintain two queues, GPU_q and CPU_q. An arriving data query will be enqueued in the shortest queue. If both queues have the same length, we have validated a tie-break heuristic that enqueues a data query in the CPU_q when its dimension, d, is smaller than 6 (since in our experiments it is probable that lower dimension datasets run faster on the CPU, as we can see in Figure 5.2), and enqueued in GPU_q otherwise.

However, as we discussed in Section 5.1.5, this highly irregular problem is solved in very disparate execution times, sometimes smaller on the CPU or on the GPU, depending on many factors. If we want to optimize the execution time, keeping busy both devices is not enough because we could want to send the data query to the optimal device, so a more elaborated strategy is necessary to feed each device with the more suitable datasets. In this regard, the Heterogeneous Earliest Finish Time [176] is an interesting alternative. This scheduling policy also takes into account the "expected" execution time of a dataset in order to feed the queues. Now, it is not the length of the queue the relevant factor, but the expected accumulated execution time of all the data queries enqueued in each queue, GPU_t and CPU_t, and the expected execution time of the arriving data query both on the GPU and CPU, t_{gpu} and t_{cpu}. This is, an arriving data query will be enqueued in the queue in which it will finish earlier. More precisely, if $GPU_t + t_{gpu} < CPU_t + t_{cpu}$ the data query will be enqueued in GPU_q, and the other way around.

This HEFT policy poses two challenges, though. First we need an accurate enough estimation of the data query execution time. We propose a heuristic (detailed in Section 5.3.1) that can infer the total execution time after sampling the time required to compute a first chunk of points of the dataset, both on the GPU and on the CPU. Considering that the skyline computation for our datasets takes more than a second, we can afford to invest around 10 ms in precomputing the first chunk of a dataset in order to estimate the total execution time. Moreover, the result obtained after this precomputation is not wasted,

but saved and never re-computed later on, so as we will see in the experimental evaluation, this strategy pays off well.

The second challenge is that estimating t_{gpu} and t_{cpu} is not possible if the GPU and the CPU are already busy processing data queries. This problem is tackled by batching the incoming data queries so that we conduct the precomputation and estimate total execution times for all the datasets in the batch. When the last data query of either the GPU or the CPU queue is launched, a new batch of data queries is sampled and HEFT is run to map them on the right queue. This not only avoids having to wait for the GPU to become idle to run HEFT, but also helps HEFT in having a farther view into the "future", which makes HEFT more profitable.

5.2.3. Fine-Grained Heterogeneous SkyFlow

We also wanted to explore a different heterogeneous approach in which the GPU and the CPU are more tightly coupled. Instead of having two different data queries being processed simultaneously on the GPU and CPU with the SkyFlow-CG implementation, the idea now is to have a single dataset partitioned so that the GPU and the CPU collaborate in the skyline computation of this data query. We call this version Fine-Grained SkyFlow, SkyFlow-FG, and the corresponding flow graph is depicted in Figure 5.5.

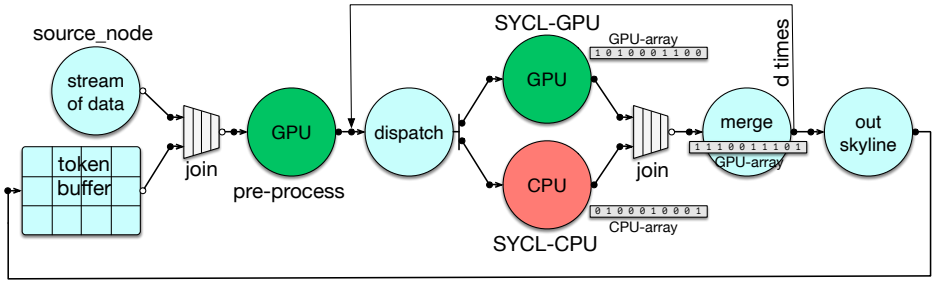


Figure 5.5: Structure of the Fine-Grained SkyFlow graph, SkyFlow-FG.

The starting point is the SYCL-GPU implementation that was described in Section 5.1.3. As we said, the code listed in Algorithm7 comprises two main stages: (1) pre-processing (lines 1-10); and (2) the main loop (lines 11- 25) that has d iterations (kernel invocations). The idea now is to efficiently distribute the

computation between the GPU and the CPU, provided that the SYCL code is portable so that the *SYCL-CPU* version can run on the CPU.

The preprocessing stage only accounts for around 10% of the execution time, and it is more than 10x faster on the GPU than on the CPU. Therefore, the slowest stage in the flow graph of Figure 5.5 is not the preprocess one which means that the preprocessing time gets hidden (one dataset is preprocessed and ready to be dispatched while the previous dataset is being computed). With all this, it is advisable to only exploit the GPU during this small fraction of the execution time.

However, the remaining 90% of the time is consumed in the main loop in which the same kernel in invoked d times (it depends on the number of dimensions). Provided that the kernel (lines 12-23 in Algorithm 7) is executed over a range of (independent) points, it is possible to partition this range so that a number of sub-ranges (or chunks) are executed by *SYCL-GPU* and the rest by *SYCL-CPU*. This implies constructing two SYCL queues, one targeting the GPU and the other attached to the CPU device (see [46] for details on constructing device queues). It also requires merging the output of the GPU partial computation with the CPU one. Both devices write in private copies of the result array (GPU_-array and CPU_array in Figure 5.5) in which dominated points are marked with ones (line 23 in Alg .7). Once both arrays have been fully written (note the `join` node after the CPU+GPU stage, instead of the `indexer` node used in Figure 5.4), the `merge` node reduces them into the GPU_array. Marked points in this summarized array are pruned from the dataset (line 24 in Algorithm7).

Moreover, this Fine-Grained implementation also requires finding the right partition of the iteration space, so that load unbalance is avoided. Again, the `join` node after the CPU+GPU stage synchronizes both devices so the slowest one sets the stage time. The dispatch, CPU+GPU execution and merge, is repeated d times which reinforces the load balance requirement. The `dispatch` node takes care of computing the right partitioning. As we will see in the next section, a dynamic partitioning will be necessary to consider that the optimal partition depends on the relative speed of each device, which can change for each dimension (out of d) and during the traversal of each dimension. This dynamic partitioning splits the iteration space into chunks of constant size. These chunks are processed on demand by the CPU and GPU devices (the device that becomes idle takes the next chunk until the iteration space is completed).

5.3. Coarse-Grained and Fine-Grained optimizations

5.3.1. Model for estimating Coarse-Grained execution times

As stated in subsection 5.2.2, the HEFT strategy requires the estimation of the execution times on the GPU and the CPU, t_{gpu} and t_{cpu}, for each dataset in an incoming batch of data queries. In this section we provide the details of the model that computes those times, which are shown in Algorithm 8. This model estimates the execution time of a dataset by profiling a small chunk of iterations both on the *OpenMP-CPU* and *SYCL-GPU* nodes.

Algorithm 8: Coarse-Grained HEFT model

Input: S=Dataset of n points and d dimensions;
Ch_{cpu}, Ch_{gpu}=CPU and GPU chunks.
Output: t_{cpu}, t_{gpu}= estimated CPU and GPU times for S.

1 (t_c, n_{cpu})=launch_OpenMP-CPU (Ch_{cpu})
2 $([t_{g_0} : t_{g_{d-1}}], [n_{g_0} : n_{g_{d-1}}], n_{gpu})$=launch_SYCL-GPU (Ch_{gpu})
3 $\lambda_c \leftarrow$ Eq. 5.6
4 $t_{cpu} \leftarrow$ Eq. 5.7
5 **foreach** *iteration* $i \in [0, d)$ **do**
6 $\quad F_i \leftarrow$ Eq. 5.9; $\quad mg_i \leftarrow$ Eq. 5.10
7 $\quad \lambda_{g_i} \leftarrow$ Eq. 5.8
8 $t_{gpu} \leftarrow$ Eq. 5.11
9 **return** (t_{gpu}, t_{cpu})

In Algorithm 8 we see that the heuristic to compute our model starts launching two chunks: Ch_{cpu} on the *OpenMP-CPU* node and Ch_{gpu} on the *SYCL-GPU* one (lines 1-2). The size of the chunk to perform the profiling is tuned at runtime. In our approach, by default it is set to 1% of the dataset size. We have found that if the reported time is around 10 ms, then the sample will typically help to provide an accurate estimation in our model. In the case that the chunk runtime is below 10 ms, then a new chunk twice the size of the previous one is launched. This process repeats until the reported processing time is above 10 ms.

It is important to note that the work computed in this profiling stage is not wasted because the points computed in these chunks are recorded and counted for the complete execution later.

For the Ch_{cpu} chunk we record the execution time, t_c, and points explored, n_{cpu}. These results are used to calculate the throughput of the CPU chunk as we see in Equation 5.6. For it, we assume a worst-case scenario in which all points of the chunk are compared when computing the OpenMP-CPU algorithm.

$$\lambda_c = \frac{(n_{cpu} \cdot (n_{cpu} - 1))/2}{t_c} \tag{5.6}$$

The estimated execution time on the CPU for dataset S, t_{cpu}, can be computed with Equation 5.7, where again we assume a worst-case scenario where all points of the dataset (n) are compared. In Section 5.4.2 we analyze the accuracy of our assumption and provide results that indicate the estimated vs measured execution times for the *OpenMP-CPU* algorithm are always within the range $\pm 10\%$ in our datasets.

$$t_{cpu} = \frac{(n \cdot (n - 1))/2}{\lambda_c} \tag{5.7}$$

For the Ch_{gpu} chunk we now record the execution time, t_{g_i}, and points explored, n_{g_i}, in each iteration i of the main loop that traverses the d dimensions of the dataset (lines 11-25 in the *SYCL-GPU* algorithm). We also record the size of the GPU chunk, n_{gpu}. In the *SYCL-GPU* algorithm, at the end of each iteration of d loop, the dominated points are removed, so fewer points enter into the next iteration. We use this information (and the time per iteration) to compute the throughput of the GPU chunk as we see in lines 5-7 in Algorithm 8. In particular, we compute the GPU throughput for each dimension i, λ_{g_i} as we see in Equation 5.8, where we assume a worst-case scenario in which all the recorded points in the corresponding dimension (n_{g_i}) are compared.

$$\lambda_{g_i} = \frac{(n_{g_i} \cdot (n_{g_i} - 1))/2}{t_{g_i}} \tag{5.8}$$

From the information collected from the GPU chunk profiling, we can compute the ratio of points filtered when going from dimension $i-1$ to dimension i. This ratio, F_i, is shown in Equation 5.9.

$$F_i = \begin{cases} \dfrac{n_{g_0}}{n_{gpu}} & \text{if } i == 0 \\ \dfrac{n_{g_i}}{n_{g_{i-1}}} & \text{otherwise} \end{cases} \tag{5.9}$$

From this ratio of filtered points, F_i, assuming an uniform distribution of the pruning of points for each dimension, we can extrapolate the number of points that will enter into each iteration of the d loop.

This number of estimated points per iteration, mg_i is computed by Equation 5.10.

$$mg_i = \begin{cases} n \cdot F_0 & \text{if } i == 0 \\ mg_{i-1} \cdot F_i & \text{otherwise} \end{cases} \tag{5.10}$$

Both the GPU throughput and estimated number of points for each dimension i, λ_{g_i} and mg_i respectively, are used to compute the estimated execution time on the GPU for dataset S, t_{gpu}, as we shown in Equation 5.11. Again, we assume a worst-case scenario where all the estimated points in each dimension (mg_i) are compared. In Section 5.4.2 we analyze the accuracy of our assumptions and provide results that indicate that the estimated vs actual execution times for the *SYCL-GPU* algorithm are within ±2% in our datasets.

$$t_{gpu} = \sum_{i=0}^{d-1} \frac{(mg_i \cdot (mg_i - 1))/2}{\lambda_{g_i}} \tag{5.11}$$

5.3.2. Strategy for the Fine-Grained partitioning

As stated in Section 5.2.3, the main kernel of the SkyFlow-FG approach, which is launched d times to process the points of the dataset, can be partitioned into sub-ranges (or chunks) of independent points, which can be computed concurrently: while one chunk is computed by the *SYCL-CPU* node, another one can be computed by the *SYCL-GPU* node. We conducted a preliminary study in which we did not partition the datasets among the devices. Instead, we launched the main kernel d times on one node (either on the CPU or on the GPU), without partitioning the points explored in each dimension, and measured the throughput per dimension d. We conducted this analysis for different configurations of our datasets (with different dataset sizes and dimensions) on our platform of reference (octa-core Intel i9-9900K CPU with integrated GPU, more details in Section 5.4.1). For instance, on Table 5.1 we show the GPU and CPU throughputs (`ThGPU`, `ThCPU`) for the NBA dataset with a configuration of 2 Million points and 7 dimensions. In addition to the throughput per dimension, we compute the total throughput and the relative speed or ratio ThGPU/ThCPU per dimension (column `Ratio`). As we see, the relative speed fluctuates for each dimension.

A similar result was reported for other datasets and configurations. Thus, the partitioning strategy must be carefully designed to consider this behavior.

In this context, a dynamic partitioning strategy seems suitable.

d	ThGPU	ThCPU	Ratio
1	$7.08 \cdot 10^{11}$	$5.88 \cdot 10^{11}$	1.20
2	$5.96 \cdot 10^{10}$	$4.25 \cdot 10^{10}$	1.40
3	$4.13 \cdot 10^{10}$	$2.80 \cdot 10^{10}$	1.47
4	$3.98 \cdot 10^{10}$	$2.34 \cdot 10^{10}$	1.70
5	$3.24 \cdot 10^{10}$	$1.77 \cdot 10^{10}$	1.83
6	$1.53 \cdot 10^{10}$	$7.24 \cdot 10^{9}$	2.11
7	$2.76 \cdot 10^{9}$	$1.58 \cdot 10^{9}$	1.75
Total	$\mathbf{6.65 \cdot 10^{11}}$	$\mathbf{5.35 \cdot 10^{11}}$	**1.24**

Table 5.1: Throughput per dimension for the main kernel on the *SYCL-GPU* and *SYCL-CPU* nodes (the higher the better). NBA dataset with $n = 2M$ and $d = 7$.

In fact, in Figure 5.6 we explore the behavior of a dynamic partitioning strategy that feeds the *SYCL-CPU* (or *SYCL-GPU*) node with chunks of points and see how the chunk size affects performance. In particular, we show the throughput on the CPU (or the GPU) device, for the first dimension traversal ($d = 1$) of the main kernel when a dynamic strategy partitions the iteration space in 10, 20, 30 and 40 chunks.

As we see in the figure, the throughput per chunk throughout the iteration space is highly variable in both devices. The same behavior is observed for the rest of dimensions of the main kernel, other datasets and configurations. This result points out that adaptive or predictive strategies based on the profiling of previous chunk samples can be misleading at this level of work granularity. So, it confirms that a dynamic partition strategy with a carefully selected chunk size must be adopted. From the figure we note that when the number of chunks is smaller, i.e. the chunk size is bigger, the measured GPU throughput tend to be higher. In fact, the GPU throughput degrades around 2% when the number of chunks doubles (i.e. the chunk size is halved). However, the CPU throughput tends to be independent of the chunk size. This result confirms that the *SkyAlign* algorithm at the core of the *SYCL-GPU* and *SYCL-CPU* implementations better exploits GPU architecture features such as coalesced memory accesses -thanks to the padding and re-packing of the main data structures-, as well as divergence

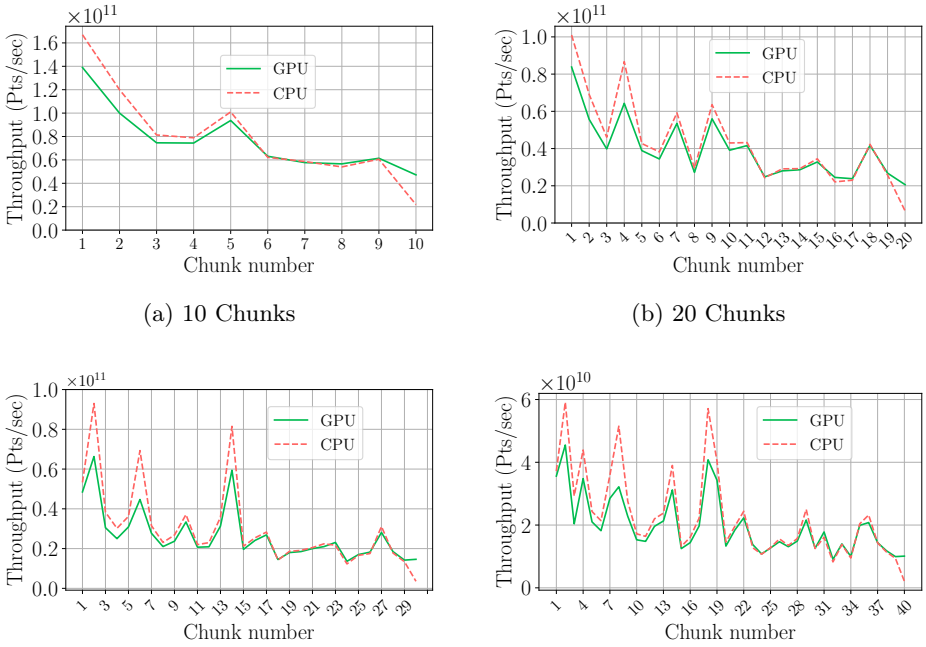


Figure 5.6: Throughput per chunk when using a dynamic partition with 10, 20, 30 and 40 chunks for the first dimension traversal of the main kernel on the *SYCL-CPU* or the *SYCL-GPU* node (the higher the better). NBA dataset with $n = 2M$ and $d = 7$.

minimization -thanks to the static mapping of the threads that in any given warp ensures that they work on a small set of aligned data blocks-.

In our SkyFlow-FG approach, the *dispatch* node (see Figure 5.5) keeps track of the sub-range of points (chunks of Chunk_size iterations) assigned to each device for each dimension d that is traversed. This node is responsible for sending chunks to the *SYCL-GPU* and *SYCL-CPU* nodes once the previous chunk computation has finished on the corresponding device. The last sub-range of the iteration space (containing at most Chunk_size iterations) is partitioned again to keep balance among devices in the final stage of the computation. In any case, when designing our heterogeneous dynamic strategy we have to seek a trade-off chunk size: big enough to fully exploit the GPU micro-architecture features and to enable a near-optimal GPU throughput, but small enough to provide a sufficient number of chunks able to feed both devices while balancing the workload at the

end of each dimension computation. Let's note in Figure 5.5 that although the chunks of iterations can be assigned asynchronously to the *SYCL-GPU*/*SYCL-CPU* nodes, after the computation of all chunks by those nodes there is a *join* node that synchronizes both devices.

In Section 5.4.3 we experimentally explore the performance of our heterogeneous dynamic strategy when selecting different chunk sizes for our datasets.

5.4. Experimental results

5.4.1. Experimental setting

The processor architecture and software details of each platform can be found in Tables 5.2 and 5.3, respectively.

Table 5.2: Platform details .

Microarchitecture	
Processor	i9-9900K 3.6 GHz
Number of cores	8
Clock Speed	3.6 GHz
Max Turbo Frequency	5.0 GHz
Main memory	32GB DDR4
Cache L3	16 MB
Litography	14 nm
Max TDP	95 W
Intel Graphics GPU	UHD Graphics 630
Number of GPU Compute Units	24
GPU Base Frequency	350 MHz
GPU Max Dynamic Frequency	1.2 GHz

CPU executions for OpenMP or SYCL algorithms use 8 threads. Times are measured using the *chrono* library [177]. The results shown in this section correspond to the median of 11 runs. Time measured do include all algorithms execution from the beginning to the end including memory allocation and release, transfers between host and device, sequential and parallel parts of the code, but do not include reading input files into CPU memory and checking results with gold execution.

Table 5.3: Software details.

Operating System	Ubuntu 20.04.3 LTS
SYCL	Intel(R) oneAPI DPC++/C++ Compiler 2021.4.0
OpenMP	g++ 9.3.0
Vectorization	AVX2
optimization flags	-O2 -fopenmp -std=c++17

We conduct the experimental evaluation of our proposals using four real datasets, widely used in the skyline research literature: House [178], NBA [179], Covertype [180], Weather [181].

- **House (HH):** This dataset [178] reports in each row one household from US and the percentage distribution of the income in different categories (e.g., rent/mortgage, electricity, etc).
- **Basketball (NBA):** This NBA basketball league dataset [179] records in each row one player's statistics from one season (e.g., points scored, free throws made, etc). While traditional statistics analysis in sports study players individually, the skyline allows to identify players that do not excels in any particular attribute but have well global performance along the set of attributes (which will correspond to a skyline point). Therefore, this dataset is one of the most widely used in skyline research.
- **Weather (WE):** The WE dataset [181] covers weather variables such as monthly precipitation, latitude, longitude and elevation in each row for different terrestrial positions around globe. Each row records 10' latitude/-longitude cell. The weather variables between the captured cells are highly dependent, since the meteorological conditions are very similar in the cells corresponding to the same regions. Thus, the WE dataset presents highly dependent attributes whose results are very interesting to be compared with other real dataset.
- **Covertype (CT):** This CT dataset [180] shows cartographic variables for 900 square meters grid cells of the Roosevelt National Forest in Colorado, USA. According skyline research, this dataset is very challenging, since many points share the extreme values. Thus, the skyline is larger in this dataset than in the others.

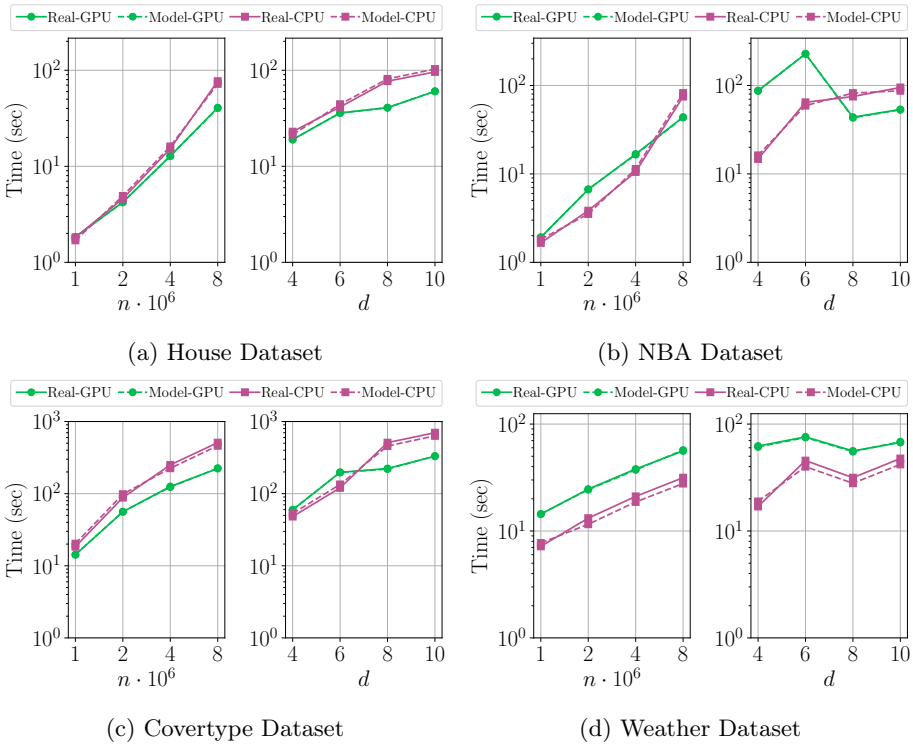
For each dataset we can define different configurations changing the number of dimensions and points. In particular, the number of dimensions can go from

4 to 10 and the datasize from 1 Million to 8 Million points. This makes a total of 224 dataset configurations, from which we will illustrate the most relevant findings in the next sections.

5.4.2. Evaluation of CG-HEFT Model accuracy

This section evaluates the accuracy of the time estimation model discussed in the Coarse-Grained HEFT Heuristic detailed in Section 5.3.1 and Algorithm 8. Let's remember that the HEFT scheduling heuristic is used by the SkyFlow-CG approach to allocate any arriving data query on the queue that guarantees to finish earlier. For that, it must be estimated the execution times for computing the skyline of the incoming dataset, both under the *SYCL-GPU* / *OpenMP-CPU* algorithm (on the GPU and CPU device, respectively). For the evaluation of our model, we run the 224 dataset configurations first on the *SYCL-GPU* node (assuming that the *OpenMP-CPU* node is not available), and then on the *OpenMP-CPU* node (assuming now that the *SYCL-GPU* one is not available). For each experiment and configuration, we record the time estimated by our model (`Est-XXX`) and the actual execution time after the skyline computation (`Meas-XXX`), both on the GPU and CPU devices. In Figure 5.7 we show a subset of these times. For each dataset, the subfigure on the left fixes $d = 8$ and changes n from $1 \cdot 10^6$ to $8 \cdot 10^6$. The right subfigure changes d from 4 to 10 for a fixed $n = 8 \cdot 10^6$.

From Figure 5.7 we see that the estimated and actual measured times are very close, both on the GPU and CPU (*SYCL-GPU* and *OpenMP-CPU* algorithms, respectively). In particular for the *SYCL-GPU* results, the difference goes from [-1.2%,0.23%] for House, [-1.6%,0.61%] for NBA, [-1.8%,0.83%] for Covertype and [-1.9%,2%] for Weather. Now, for the *OpenMP-CPU* experiments the range goes from [-6.93%,7.2%] for House, [-8.28%,9.72%] for NBA, [-10.18%,10.45%] for Covertype and [-10.3%,10.5%] for Weather. A negative value means the model overestimates the actual measured time, while a positive one indicates that the model underestimates it. Although the accuracy is slightly worse for *OpenMP-CPU* compared to *SYCL-GPU*, our model is still accurate enough for the CG-HEFT scheduling heuristic. We base this claim in the fact that, for any given arriving data query the difference between the actual execution times on each device is much higher (from 1.5x to 4x) than the $\pm$ 10% of inaccuracy incurred by the model when making the decision to enqueue on one device. In other words, our model always selects the appropriate queue.



(a) House Dataset

(b) NBA Dataset

(c) Covertype Dataset

(d) Weather Dataset

Figure 5.7: Estimated vs actual measured times for the *SYCL-GPU* and *OpenMP-CPU* algorithms and four datasets (the lower the better).

5.4.3. Evaluation of the partition strategy in Fine-Grained Heterogeneous SkyFlow

In this section we analyze the performance of the heterogeneous dynamic partition strategy proposed in Section 5.3.2 whose goal is to find the near-optimal workload assigned to the *SYCL-GPU* and *SYCL-CPU* nodes in order to optimize the throughput in the SkyFlow-FG approach. As discussed in the mentioned section, this strategy asynchronously assigns chunks of iterations to the GPU and CPU devices for each traversal d of the main loop. The critical design issue is to find a chunk size big enough to ensure a near-optimal GPU throughput, but small enough to provide a sufficient number of chunks able to feed both devices while balancing the workload at the end of each iteration of the d loop. In Figure 5.6 we showed the evolution of the throughput per chunk on each

device when setting different number of chunks (and therefore different chunk sizes) in a dynamic partitioning on a configuration of the NBA dataset. Now in Figure 5.8, for the same dataset configuration, we show the average throughput when running the dynamic partitioning only on the *SYCL-CPU* node (`ThCPU`), only on the *SYCL-GPU* node (`ThGPU`), and compare them with the average throughput when running the heterogeneous dynamic partitioning on the *SYLC-CPU+SYCL-GPU* nodes (`ThCPU+GPU`). Also, we depict an ideal throughput (`Ideal`) computed as the aggregation of the throughputs on the CPU and GPU without partitioning, so the partitioning overhead and load unbalance between devices is factored out.

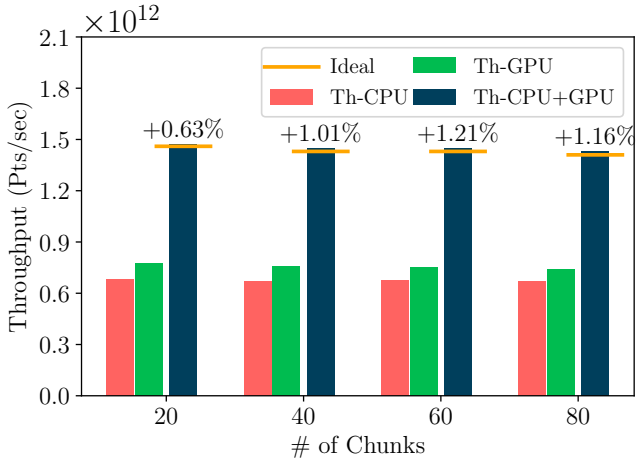


Figure 5.8: Average throughput for the dynamic partitioning strategy in the SkyFlow-FG approach and different number of chunks (the higher the better). Device-only executions (`ThCPU`, `ThGPU`) are compared with heterogeneous execution (`ThCPU+GPU`) and ideal throughput (`Ideal`). The values represent the percentage of performance degradation of the heterogeneous CPU+GPU execution with respect to the ideal throughput (the lower the better). NBA dataset with 2M points and 7 dimensions.

As explained in Section 5.3.2, whereas the `ThCPU` is constant regardless the number of chunks, the `ThGPU` tends to slightly degrade when increasing the number of chunks (i.e., when reducing the chunk size). The GPU throughput degrades slightly: around 2% when the number of chunks is doubled. Interestingly, the heterogeneous `ThCPU+GPU` increases from 10 to 20 chunks, and then it degrades slightly from 30 onward (less than 1%). The values we see in the fig-

ure (the percentage of performance degradation of the heterogeneous execution with respect to the ideal throughput) helps to quantify, in part, the impact of load unbalance on the heterogeneous performance, because smaller chunks tend to minimize the unbalance. As we see, when the number of chunks is small (10, i.e. bigger chunks), load unbalance is the main factor that explains the 5.7% of performance degradation. Increasing the number of chunks to 20 (decreasing chunk size) reduces load unbalance to 1.95% (the sweet spot for this dataset configuration). However, from 30 chunks onward we see again degradation of the heterogeneous throughput compared to the ideal: now the minimization of the load balance is not compensated by the degradation of the GPU throughput due to smaller chunks. We conducted an exhaustive exploration of the optimal number of chunks (chunk size) for each dataset and configuration, and the optimal values that were found are used for the results of the SkyFlow-FG approach that we present in the next section.

5.4.4. Evaluation of heterogeneous SkyFlow approaches

In this section we present performance results of the SkyFlow approaches that we introduce in Section 5.2. We measure the performance when streaming 100 data queries and record the median of 11 runs. One stream of data queries consist of mixed configurations (dataset size and dimensions) of the same dataset. As explained in Section 5.1.5, the performance of the skyline computation of a data query is highly irregular, heavily depending on the dataset configuration, algorithm and device. Thus, to thoroughly study the efficiency of our proposals, we evaluate two streaming scenarios: `Random (R)` and `Unbalanced (U)`. Whereas the `Random` scenario contains a random distribution of data queries, the `Unbalanced` scenario contains only data queries that run faster using *SYCL-GPU* than *OpenMP-CPU*. The second column of Table 5.4 provides details of the number of data queries that are faster with *SYCL-GPU* and with *OpenMP-CPU* under the `R` scenario for each dataset. The table also reports the mean times (in seconds) that the baselines SkyFlow-GPU and SkyFlow-CPU archive for the `R` and `U` stream scenarios.

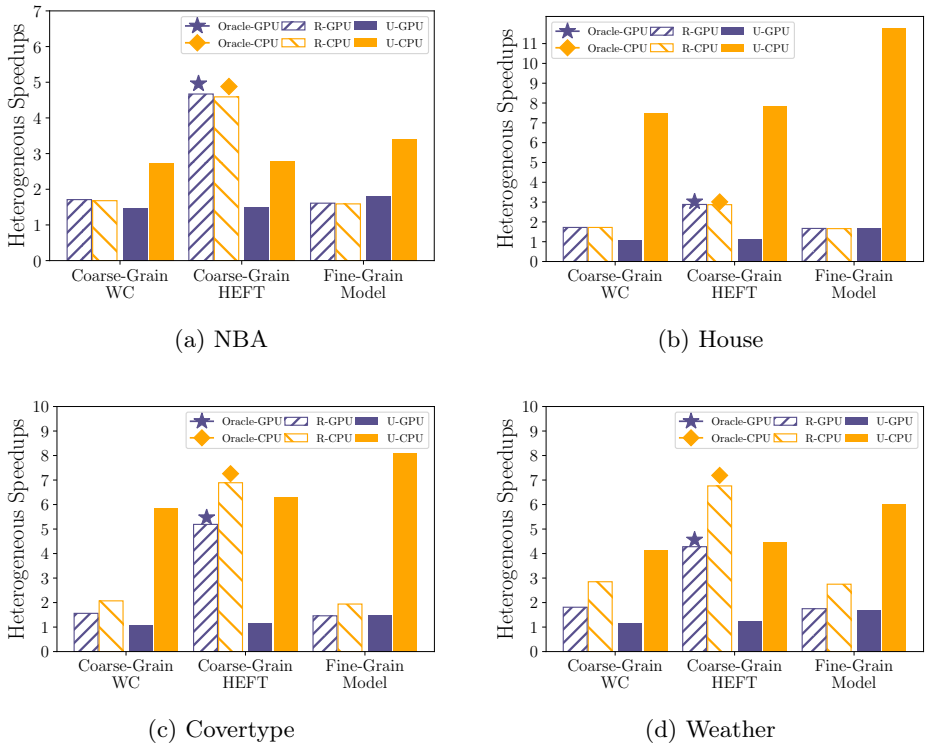
The performance improvement of our heterogeneous proposals: SkyFlow-CG under WC scheduling (`SkyFlow-CG WC`), SkyFlow-CG under HEFT scheduling (`SkyFlow-CG WC`) and SkyFlow-FG under optimal dynamic partitioning (`SkyFlow-FG`), are presented in Figure 5.9 for the two streaming scenarios (`R` -patterned bars- and `U` -solid bars-) in our four datasets. The performance improvement is presented as the speedup of each heterogeneous proposal vs. the baseline SkyFlow-GPU and the baseline SkyFlow-CPU (see Figure 5.3), for both

Table 5.4: Data queries mix for scenario R; Mean times (sec.) for baselines SkyFlow-GPU and SkyFlow-CPU (the lower the better) for scenarios R and U in our four datasets.

Dataset	**R: GPU-CPU queries**	**R: SkyFlow-GPU time**	**R: SkyFlow-CPU time**	**U: SkyFlow-GPU time**	**U: SkyFlow-CPU time**
NBA	44-56	2596.70	2464.87	4659.91	8719.60
House	38-62	894.04	890.81	701.56	4841.28
Covertype	13-87	3041.16	4032.88	4495.52	24311.40
Weather	22-78	1927.74	3040.70	2100.33	7438.54

streaming scenarios (named R−GPU, U−GPU and R−CPU, U−CPU respectively). Evaluating the performance improvement of the heterogeneous implementations against the homogeneous baselines helps us to quantify the gain of heterogeneous implementations compared to single-devices ones in complex streaming scenarios.

As we see in Figure 5.9, our heterogeneous approaches always outperform SkyFlow-GPU and SkyFlow-CPU baselines in the two streaming scenarios and four datasets. In fact, they outperform GPU and CPU baselines up to 5.19x and 6.86x, respectively. This result tells us that exploiting both devices with our heterogeneous solutions is usually more profitable than using just one device. Even if the device selected for the arriving data query is not the optimal one (CG approaches), or even if we partition the data points among devices (FG approach). For the U scenario, all the data queries are faster on the *SYCL-GPU* node, so the times for the baseline SkyFlow-CPU always take longer than for the SkyFlow-GPU (see the last two columns in Table 5.4). Thus, any heterogeneous approach that considers the GPU for this stream of data queries will show an important speedup when compared to SkyFlow-CPU (U−CPU) vs the speedup that we obtain when compared to SkyFlow-GPU (U−GPU) (see yellow solid bars vs blue solid bars). Regarding the R scenario (the patterned bards), the speedup against baseline SkyFlow-CPU (R−CPU) and SkyFlow-GPU (R−GPU) tend to be similar for NBA and House datasets, because the stream of data queries takes similar time in both datasets, while for Covertype and Weather datasets the speedup against SkyFlow-CPU is higher than the speedup against SkyFlow-GPU, because the times for SkyFlow-CPU take longer than for the SkyFlow-GPU in these cases (see third and fourth columns in Table 5.4). In any case, in the next subsections we discuss the main findings for our heterogeneous approaches.



(a) NBA (b) House

(c) Covertype (d) Weather

Figure 5.9: Performance improvement of SkyFlow proposals for the four datasets and two streaming scenarios: `R` for random and `U` for unbalanced. The improvement is computed as a speedup vs the baseline SkyFlow-GPU (`-GPU`) and the baseline SkyFlow-CPU (`-CPU`). Oracle represents the optimal scheduling of queries for the SkyFlow-CG approach, which has been evaluated offline. The higher the better.

5.4.4.1. Analysis of Coarse-Grained heterogeneous scheduling strategies

The SkyFlow-CG is a hybrid approach that considers two skyline algorithmic implementations that are optimal for each one of the two devices we target in this work: *SYCL-GPU* and *OpenMP-CPU*. As illustrated in Section 5.1.5, in our platform the optimal algorithm-device depends on characteristics of the arriving data query. The scheduling strategies proposed in Section 5.2.2, WC and HEFT, allocate any arriving data query either on the *SYCL-GPU* or the *OpenMP-CPU* queue following two different goals. The WC policy tries to keep the length of the queues equalized (same number of pending tasks) independently of which

algorithm-device is best suited for the arriving query. By contrast, the HEFT strategy enqueues the incoming query in the queue in which it will finish earlier. While WC has minimum scheduling overhead and ensures that the devices are not idle if there are data queries ready, the HEFT strategy tries to optimize the system throughput. For the HEFT strategy and for the two streaming scenarios and the four datasets evaluated in this work, we have found that a batching size of 5 data queries (to perform the estimation of the expected times for the data queries in the batch) provides a good trade-off between the overhead due to the precomputation required for the times estimations and the likelihood that one of the devices becomes idle when one of the queues runs out of queries.

From Figure 5.9 we see that HEFT strategy always outperforms WC in all scenarios and datasets. Although WC scheduling minimizes the idle time on each device, it introduces a scheduling inefficiency by enqueueing queries in the non-optimal device. This inefficiency has a larger impact in the `R` scenario, where HEFT outperforms WC by 2.73x, 1.67x, 3.32x and 2.36 for NBA, House, Covertype and Weather, respectively. However, in the `U` scenario, HEFT outperforms WC by 1.02x, 1.04x, 1.08x and 1.07x, respectively, what demonstrates that even for non favorable situations HEFT still makes better scheduling decisions than WC. Another remarkable finding is that in the HEFT experiments we measure a time standard-deviation that goes from 0.17% to 0.47%, while in the WC runs it goes from 0.85% to 6.5%. These results point to the fact that HEFT also produces more stable executions.

Interestingly, in the most unfavorable scenario, that is `U` streaming, WC achieves improvements between 1.08x and 1.46x when compared to the optimal baseline SkyFlow-GPU. This corroborates the fact that from the point of view of the whole system performance, trying to keep both devices busy is still more beneficial that leaving the CPU idle, even if the CPU runs queries for which it is not the best device.

To measure the efficiency of our HEFT scheduling heuristic, we compare its performance with an Oracle approach that first computes offline the optimal device for each data query and uses this information to enqueue the query. Oracle also avoids the overhead of precomputing the expected execution time on any device for any incoming data query. Thus, Oracle represents the ideal peak performance for any CG scheduling policy. The results show that the performance of HEFT is below Oracle in 6.08%, 4.84%, 5.14%, 5.14% for NBA, House, Covertype and Weather, respectively. As we see, our scheduling heuristic introduces a low overhead while ensuring a near-optimal scheduling of data queries.

5.4.4.2. Study of Coarse-Grained vs Fine-Grained approaches

In this section we compare the performance of the SkyFlow-FG approach with the SkyFlow-CG HEFT, since the last one always outperforms SkyFlow-CG WC. Let's recall that in SkyFlow-FG, the workload of each data query is dynamically partitioned between the *SYCL-GPU* and *SYCL-CPU* nodes, and that the size (and number) of data chunks must be carefully selected, as we discussed in subsection 5.4.3. From Figure 5.9 we see that SkyFlow-CG HEFT always outperforms SkyFlow-FG in the `R` scenario, because SkyFlow-CG HEFT takes advantage of the better adaptation of each specific query to the most suitable algorithm-device. However, fine-grained work partition in SkyFlow-FG does not pay off for the execution of queries with suboptimal performance under the SYCL algorithm.

On the other hand, in the `U` scenario we get some interesting results. In this case all the queries run faster under the SYCL algorithm. SkyFlow-FG will achieve optimal performance for all the queries, while SkyFlow-CG HEFT will degrade performance when executing queries in the *OpenMP-CPU* node. In this scenario SkyFlow-FG outperforms SkyFlow-CG HEFT by 1.22x, 1.5x, 1.27x and 1.37x for NBA, House, Covertype and Weather, respectively. In other words, when the incoming data queries run faster on the GPU device (SYCL) it can be advantageous to exploit a dynamic fine grain partition of the workload of each query between the GPU and CPU devices.

5.5. Conclusions

The skyline is an optimization operator widely used for multi-criteria decision making. It allows minimizing an n-dimensional dataset into its smallest subset. In this work we present SkyFlow, the first heterogeneous CPU+GPU graph-based engine for skyline computation on a stream of data queries. Two data flow approaches, Coarse-grained and Fine-grained, have been proposed for different streaming scenarios. Coarse-grained aims to keep in parallel the computation of two queries using a hybrid solution with two state-of-the-art skyline algorithms: one optimized for CPU and another for GPU. We also propose a model to estimate at runtime the computation time of any arriving data query. This estimation is used by a heuristic to schedule the data query on the device queue in which it will finish earlier. On the other hand, Fine-grained splits one query computation between CPU and GPU. Our experimental results show that in our streaming scenarios and datasets, our heterogeneous CPU+GPU approaches

always outperform previous only-CPU and only-GPU state-of-the-art implementations up to 6.86x and 5.19x, respectively, and they fall below 6% of ideal peak performance at most. We also evaluate the suitability of Coarse-grained vs Fine-Grained in two different streaming scenarios: random and unbalanced.

Summarizing, the main contributions of this chapter are:

- We contribute, to the best of our knowledge, with the first heterogeneous implementation for skyline computation. We port from CUDA to oneAPI the state-of-the-art algorithm, splitting the computation between CPU and GPU. We also contribute with a dynamic partitioning heuristic to optimize performance between devices.
- We propose two heterogeneous implementations for skyline computation over a streaming of data queries: Coarse-grained and Fine-grained. Coarse-grained keeps two skyline computation in parallel, one per device, while in Fine-grained a single skyline computation is split between CPU and GPU. We validate the suitability of each implementation for different streaming scenarios.
- We present two policies for balancing workloads between devices in the Coarse-grained approach. Each device has a queue assigned for enqueueing queries. The first approach keeps devices busy offloading queries in the shortest queue. The second approach estimates the execution time for the input query on each device. To such end, we develop a model that estimates the execution time of an input query with negligible overhead, by sampling a small chunks of points at runtime. That estimated time is used to balance the workload considering the accumulated estimated execution times in each queue.

6 Concluding Remarks

This chapter summarises the findings of this thesis. We propose several runtime strategies for optimizing two irregular massive data applications in heterogeneous architectures: the Matrix Profile and the skyline computation over a strem of data queries. Our goal is exploring programming models that allow us to minimize the user's programming effort while getting the most out of these applications performance when ported to accelerator-based heterogeneous platforms.

Section 6.1 summarizes the contributions presented in the publications derived from the central chapters of this thesis. Section 6.2 details the limitations encountered after the completion of the work. Section 6.3 focuses on the future lines of research that would be the natural continuation of this thesis.

6.1. Contributions

Chapter 3 explores different alternatives to optimally map the Matrix Profile algorithm onto commodity processors featuring several CPU cores and an integrated GPU. We first validated that in homogeneous scenarios (only CPU cores) a dynamic partitioning of the iteration space (diagonals) based on oneTBB and work stealing outperforms (by up to 11.3%) an ideal static distribution that evenly distributes, a priori, the number of elements.

A further step to better exploit the heterogeneous chips consists in also offloading part of the computational burden to the GPU. With this in mind, we propose HetMP, based on oneTBB, an extension of a previous library developed

by our research group, which offers an easy abstraction layer and provides several scheduling strategies that exploit the Shared Virtual Memory (SVM) feature available in OpenCL 2.1, as well as different solutions to the workload balance among devices. We develop and analyze two versions of the Matrix Profile kernel, NonAtomic (faster but unsafe) and Atomic (slower but precise), concluding that the Atomic version only incurs in up to 8.9% of performance degradation w.r.t. the NonAtomic one for the GPU only execution. Three alternatives to distribute the workload between the CPU and the GPU have been evaluated (*Static*, *Dynamic* and *LogFit*), finding that the most productive one is *LogFit* when an exact solution is required. However, if the magnitude to minimize is the energy consumption, offloading all the computation to the GPU is advisable. For future work we plan to extend the heterogeneous scheduler so that it can be instructed to minimize energy or a performance/energy tradeoff, instead of always seeking maximum performance.

Chapter 3 has been elaborated on the basis of the following publications:

Time Series collaborative execution on CPU + GPU chips.
Jose Carlos Romero, Angeles Navarro, Andrés Rodríguez, Rafael Asenjo and Murray Cole
In *HPC-Europa3 Transnational Access Meeting (TAM)*, Edinburgh, Scotland, October 2018. (International Conference).

Time Series heterogeneous Co-execution on CPU + GPU chips.
Jose Carlos Romero, Angeles Navarro, Andrés Rodríguez, Rafael Asenjo and Murray Cole
In *19th International Conference Computational and Mathematical Methods in Science and Engineering (CMMSE)*, Cadiz, Spain, July 2019. (International Conference).

ScrimpCo: scalable matrix profile on commodity heterogeneous processors.
Jose Carlos Romero, Antonio Vilches, Angeles Navarro, Andrés Rodríguez and Rafael Asenjo.
In *The Journal of Supercomputing volume 76, pages 9189–9210* , February 2020. DOI: 10.1007/s11227-020-03199-w. JCR T1/Q2 Journal. Category:

Computer Science, Theory and Methods. Ranking category 33/110. Impact Factor: 2.474).

Chapter 4 proposes a novel hierarchical scheduler named *Fastfit*, to efficiently balance the workload in a heterogeneous FPGA-based system - that includes banks of HBM[1] - while ensuring near-optimal throughput with minimal runtime overhead, and we have used the Matrix Profile to illustrate its applicability. *Fastfit* is a system-level scheduler based on an analytical model of the FPGA pipeline IPs that helps us to find the FPGA chunk size that guarantees near-optimal FPGA throughput, and from that, the CPU chunk size that ensures load balance among devices. Besides, *Fastfit* includes a device-level scheduler that provides an effective partition of the FPGA chunk into sub-chunks for each FPGA IP.

Through exhaustive evaluation, we validate the accuracy of our models and the optimality of *Fastfit* for getting the near-optimal FPGA chunk size, finding that our model prediction is within the 97%-99% of the actual measured best throughput. We also compare different strategies for performing the device-level partition of the FPGA chunk among IPs, finding that the *Balanced* strategy that is aware of the triangular geometry of the problem improves the performance of a naive Block one by 16.45%. We also find that a simple model of the HBM usage bandwidth and the sharing of banks among IPs allow us to set the minimum number of active banks that ensure the maximum aggregated memory bandwidth while reducing power consumption.

We compare our proposed scheduler with previous scheduling strategies (*Static*, *Dynamic* and *Logfit*) and we demonstrate that our new scheduler improves all of them in terms of performance. Moreover, *Fastfit* is 4.68% better than *Logfit*, a previous state-of-the-art adaptive scheduler that finds the near-optimal chunk size for each device without the need of offline profiling. The reason of that improvement is that our new proposal avoids the logarithmic fitting overheads of *Logfit*. In fact, *Fastfit* is only 0.6% away from the ideal heterogeneous execution. However, if our goal is to minimize energy consumption, offloading all the workload to the FPGA is the best choice. Using only FPGA reduces in -57.88% the energy consumption and improves in 36.66% the energy efficiency when compared to *Static* -the second best scheduler regarding energy metrics-.

Chapter 4 has been elaborated on the basis of the following publication:

[1] HBM=*High Memory Bandwidth*

Efficient heterogeneous Matrix Profile on a CPU + High Performance FPGA with integrated HBM.
Jose Carlos Romero, Angeles Navarro, Antonio Vilches, Andrés Rodríguez, Francisco Corbera and Rafael Asenjo.
In *Future Generation Computer Systems, 125, pages 10-23*, December 2021. DOI: 10.1016/j.future.2021.06.025. JCR T1/Q1 Journal. Category: Computer Science, Theory and Methods. Ranking category 7/110. Impact Factor: 7.187.

Chapter 5 tackles the problem of computing the skyline operator over a stream of independent data queries targeting a heterogeneous architecture comprised of a multi-core CPU and an integrated GPU. For it, we propose a heterogeneous graph-based engine, called SkyFlow to efficiently schedule the data queries between the devices. We propose two approaches that adapt to different streaming scenarios: Coarse-grained (SkyFlow-CG) and Fine-grained (SkyFlow-FG).

SkyFlow-CG computes concurrently one query per device, using a hybrid approach: each device runs the algorithm best suited to the specific features of the corresponding device. For our platform this means that the CPU runs the state-of-the-art *Hybrid* algorithm - based on an OpenMP implementation-, while GPU executes the state-of-the-art *SkyAlign* - based on a SYCL implementation, novel in this work-. Although both algorithms exploit work efficiency by reducing the number of dominance tests required during the skyline computation, for the real datasets evaluated in our work we have found that on our system, some of the queries perform better under *OpenMP-CPU*, while others under *SYCL-GPU*.

For the SkyFlow-CG approach we consider two scheduling strategies: Work Conserving (SkyFlow-CG WC) and Heterogeneous Earliest Finish Time (SkyFlow-CG HEFT). While WC aims to keep all devices busy by enqueueing any arriving data query on the shortest device queue, the HEFT strategy tries to optimize the system throughput by enqueuing the incoming query on the device queue in which it will finish earlier. HEFT requires estimating at runtime the computation time of an arriving query on each device. For it, in this chapter we introduce a novel model that is based on an initial sampling of some points of the dataset, executed under *SYCL-GPU* and *OpenMP-CPU*. Through exhaustive evaluation we have found that the inaccuracy incurred by our model is within $\pm$ 10% of actual skyline computation times, and it is always smaller than the time difference between algorithm-devices. As a result, our model always selects the optimal device. In any case, in our evaluation of the scheduling strategies, HEFT always outperforms WC in all streaming scenarios and datasets. In particular, the

HEFT strategy outperforms WC up to 3.32x in random scenarios that contain a random mix of queries well suited for each algorithm-device. Moreover, when we compare the performance of our HEFT heuristic against an Oracle strategy that computes offline the optimal device for each query, we find that HEFT only degrades Oracle peak performance by 6% at most. This corroborates that our model-based heuristic introduces a low overhead while ensures the optimality of the scheduling.

Secondly, the SkyFlow-FG approach dynamically partitions the workload of each arriving data query between the *SYCL-GPU* and *SYCL-CPU* nodes. Through careful selection of the size (and number) of data chunks sent to each device, we have found that this fine-grained partition strategy is beneficial in a streaming scenario where the majority of data queries run faster under the SYCL algorithm. In this scenario SkyFlow-FG outperforms SkyFlow-CG HEFT up to 1.5x.

Chapter 5 has been elaborated on the basis of the following publication:

SkyFlow: Heterogeneous Streaming for Skyline computation using FlowGraph and oneAPI.
Jose Carlos Romero, Felipe Muñoz, Antonio Vilches, Angeles Navarro, Andrés Rodríguez and Rafael Asenjo
In *VI Congreso Nacional de Informática (CEDI)*, Málaga, Spain, Sep 2021. (National Conference).

SkyFlow: Heterogeneous Streaming for Skyline computation using FlowGraph and SYCL.
Jose Carlos Romero, Angeles Navarro, Andrés Rodríguez and Rafael Asenjo
In *Future Generation Computer Systems (under review)*, June 2022. JCR T1/Q1 Journal. Category: Computer Science, Theory and Methods. Ranking category 7/110. Impact Factor: 7.187.

6.1.1. Answer to Research Questions

Here we answer the Research Questions introduced in Chapter 1:

RQ #1: Is it possible to develop an optimal heterogeneous implementation for the state-of-the-art Tiem Series algorithm?

The answer to this RQ is positive, as stated in the work described on Chapters 3 and 4. Chapter 3 presents the first heterogeneous implementation for the state-of-the-art Time Series algorithm on a CPU + GPU architecture and Chapter 4 extends this contribution to a CPU + FPGA architecture. We evaluate the efficiency of three heterogeneous schedulers (*Static*, *Dynamic* and *LogFit*) for our heterogeneous implementations, and propose a new scheduler specifically designed for CPU + FPGA architectures, *Fastfit*.

RQ #2: Can FPGA with HBM capabilities be an efficient accelerator for Time Eeries computation? And more specifically: Is it possible to develop a scheduler which leverage such accelerator in an heterogeneous implementation for the state-of-the-art Time Series algorithm?

The work developed in Chapter 4 answers this RQ positively. The experimental evaluation shows that an FPGA with HBM capabilities is a competitive accelerator for Time Series computation not only considering performance but also energy consumption and energy efficiency. Furthermore, we also develop a methodology based on a analytical model to optimize the memory bandwidth usage of the HBM banks. More specifically, it is also possible to develop an scheduler to leverage the FPGA in an heterogeneous implementation. Our experimental results show that *Fastfit*, our proposed scheduler, achieves up to to 99.4% of ideal performance.

RQ #3: Is it possible to develop an optimal heterogeneous implementation for the state-of-the-art skyline algorithm and a model to optimize its performance?

The answer to this RQ is positive, as can be seen in the work developed in Chapter 5. In the chapter we contribute, to the best of our knowledge, with the first heterogeneous CPU + GPU implementation, based on the only-GPU state-of-the-art algorithm, for skyline computation. Our implementation is optimized with a dynamic partitioning to balance the workload between devices.

We propose two heterogeneous implementations for skyline computation over a stream of data queries: Coarse-grained, using only-CPU and only-GPU state-of-the-art implementations and Fine-grained, based on the only-GPU state-of-the-art algorithm. Coarse-grained keeps two skyline computations in parallel, one per device, while in Fine-grained a single skyline computation is split between the CPU and the GPU. Our heterogeneous CPU+GPU approaches always outperform previous only-CPU and only-GPU state-of-the-art implementations up to 6.86x and 5.19x, respectively, and they fall below 6% of ideal peak performance at most.

RQ #4: In the context of the skyline computation, can a multi-algorithm scheduler be developed for a continuous streaming of datasets that maps the input received into the best-suited device to optimize the overall system performance?

Continuing with the contributions of Chapter 5, the answer to this RQ is also positive. We present a multi-algorithm scheduler based on the Heterogeneous Earliest Finish Time (HEFT) strategy that keeps a queue per device. The scheduler estimates the execution time for the input query on each device. To such end, we develop a model that, at runtime takes small chunks of points from a dataset and from that extrapolates and estimates the execution time of the whole dataset with negligible overhead. That estimated time is used to balance the workload considering the accumulated estimated execution times on each queue. We validate the suitability of the scheduler for different streaming scenarios.

6.2. Limitations

Whereas this thesis provides several contributions, it also has some limitations. The first limitation is related to code portability. For the different schedulers proposed in Chapter3 and 4, we require that the user provides two versions of the kernel, one for the CPU cores and another for the GPU/FPGA accelerator. However, it would be beneficial to express a unique version of the kernel that is compiled and run on each computational device within the system. In this regard, we are working on porting the existing OpenCL code to DPC++, thanks to the SYCL higher-level abstraction layer that supports all of OpenCL features. DPC++ enables the convenience, productivity, and flexibility of single-source C++. With the kernel code embedded in the host code, programmers gain the simplicity of coding and compilers gain the ability to analyze and optimize across the entire program regardless of the device on which the code is to be run. DPC++ accomplishes this through single-source multiple-compiler passes (SMCP). With SMCP, a single source file is parsed by different compilers for different target devices generating different device binaries. In many cases, those binaries are combined into a single executable. In this way, we will be able to target different devices with the same source code, such as the work conducted in Chapter 5.

Another limitation of this thesis is that our scheduling strategies consider our application as the only one running on the system. However, as it is so often the case in heterogeneous architectures, several applications may be running at the same time. As our analytical models and schedulers make this simple assumption, they will overestimate their performance predictions, so the workload between

devices will not be correctly balanced, which will have a negative impact on the performance. More research should be conducted in these multi-workload scenarios, where our dynamic and adaptive approaches could be tuned to include the effects of competition for resources.

6.3. Future work

To conclude this thesis, we want to propose future research lines related with our work.

- As mentioned in Chapter 3, our heterogeneous scheduler is focused always in seeking maximum performance. However, other requirements could be used as criterion of optimality. We could extend our heterogeneous scheduler so it can also target the optimization of other metrics, such as energy consumption or a performance/energy tradeoff.

- Regarding Chapter 4, the accelerator studied there, an FPGA with support for HBM, uses floating-point arithmetic to perform operations. We plan to explore the use of a fixed-point arithmetic to optimize the FPGA kernels. Fixed-point arithmetic not only allows to perform faster operations than floating-point ones, but also needs less hardware. This way the use of fixed-point arithmetic would improve the FPGA implementation performance and energy consumption. Moreover, recently the FPGA vendor has released an BSP to support oneAPI in an FPGA with HMB. Hence, a code migration from OpenCL to DPC++ can be tackled to enable the HBM memory banks exploitation using SYCL. Also, our hierarchical scheduler, Fastfit, could be extended to target others accelerators available in the system, such as an integrated/discrete GPU. This way, the scheduler should include this third device in the scheduling strategy. Finally, the scheduler could also be extended to target other metrics, such as power consumption or performance/power ratio, which are of particular interest when using a low-power device such as an FPGA.

- Lastly, in Chapter 5, we plan to explore the use of FPGA as an additional accelerator device when computing the skyline operator over a stream of queries. An FPGA with HBM banks could improve the performance of existing FPGA implementations for skyline computations, a topic that has not been explored. The addition of this new accelerator could lead to combinations at runtime of Coarse-grained (SkyFlow-CG) and Fine-grained

(SkyFlow-FG) data queries computations. For instance, two skylines could be executing in parallel, one of them on the FPGA fully exploiting the HBM capabilities - following the SkyFlow-CG approach-, and the second one on the CPU + GPU - following the SkyFlow-FG approach -.

Appendix A
Resumen en español

Optimización de aplicaciones de análisis masivo de datos en arquitecturas heterogéneas

Desde el cambio de paradigma de las arquitecturas mononúcleo a las arquitecturas multinúcleo, la arquitectura de los procesadores no ha dejado de progresar. Hoy en día, en el diseño de nuevas arquitecturas no sólo el rendimiento se ha convertido en un requerimiento principal a tener en cuenta, sino también el consumo de energía. En los últimos años, las arquitecturas heterogéneas se han impuesto en todos los ámbitos de la industria: desde los superordenadores hasta las plataformas móviles y los sistemas de IoT. El desarrollo de dispositivos especializados (GPU, NPU, TPU, FPGA) que colaboran con las CPUs en SoC han dado lugar a sistemas heterogéneos que han permitido no sólo una mejora de la eficiencia energética respecto a los sistemas multinúcleo tradicionales, sino también del rendimiento. En este trabajo se discuten los retos y fortalezas de utilizar arquitecturas heterogéneas para optimizar el rendimiento y el consumo de energía para resolver aplicaciones clave de análisis masivo de datos.

A.1. Motivación

Según la ley de Moore [1], en los últimos cincuenta años el número de transistores por chip se ha duplicado cada dos años. Este hecho ha ido acompañado de un aumento de la frecuencia y del consumo de energía que a mediados de la década de los 2000 se hizo insostenible. Este límite llevó a estabilizar la frecuencia para controlar la densidad de potencia consumida en el chip, apareciendo como primera solución la arquitectura multinúcleo. Por lo que la industria cambió su estrategia para seguir aumentando el rendimiento, pasando de aumentar la frecuencia del núcleo a añadir más núcleos en un chip. Esta innovación supuso un reto para los desarrolladores a la hora de aprovechar las nuevas arquitecturas: esto suponía adaptar los códigos para aprovechar las capacidades multinúcleo de los nuevos chips [2].

En la última década se ha producido una revolución en el diseño de la arquitectura de los procesadores, ya que los chips han pasado de los sistemas multinúcleo a las arquitecturas heterogéneas. Ahora los chips no contienen sólo réplicas de un mismo núcleo (sistemas multinúcleo), sino dispositivos especializados que colaboran con la CPU en un SoC. Ejemplos de estos dispositivos especializados son: GPU, FPGA, DSP, NPU and TPU.

La computación heterogénea es el paradigma por el que una aplicación se optimiza para que la carga computacional (típicamente tareas) se reparta entre los distintos dispositivos que integran una arquitectura heterogénea. Esta optimización puede resultar compleja ya que la orquestación de las tareas entre los dispositivos tiene que tener en cuenta la capacidad de cómputo de cada uno, los mecanismos de sincronización entre ellos y la jerarquía de memoria. La idea clave de este paradigma es asignar las tareas que mejor se adaptan a un dispositivo concreto, optimizando el código para cada dispositivo utilizando funcionalidades que el hardware o el modelo de programación exponen al programador.

En esta tesis nos centramos en las arquitecturas heterogéneas CPU + acelerador, como por ejemplo GPU o FPGA. Estas plataformas las utilizamos para resolver problemas reales que representan aplicaciones de análisis masivo de datos, y presentamos estrategias que en tiempo de ejecución optimizan el uso de los recursos en estas arquitecturas. Una característica de las aplicaciones que estudiamos en este trabajo es que son ïrregulares". Los problemas irregulares son aquellos en los que la distribución de la carga computacional varía a lo largo del espacio de iteraciones que definen el núcleo de la aplicación. En algunos problemas la irregularidad puede ser regularizada. Esta aproximación se consigue utilizando estrategias que permiten modelar la carga y dinámicamente en tiempo

de ejecución encontrar la mejor distribución de trabajo entre la CPU multinúcleo y el acelerador . En otros problemas, sin embargo, la distribución de la carga de trabajo es completamente impredecible e incluso puede cambiar dependiendo de los resultados anteriores o de la entrada recibida. Una posible solución en este caso es buscar una partición dinámica y adaptativa de la carga de trabajo para mantener el balanceo entre los dispositivos en todo momento.

En la literatura se han propuesto varios enfoques para solucionar el problema del balanceo de la carga de trabajo. Algunos de ellos proponen una fase de entrenamiento previa a la ejecución para averiguar la mejor distribución de la carga de trabajo entre los dispositivos [14, 15, 16, 17]. Trabajos recientes han propuesto soluciones para la optimización de aplicaciones irregulares utilizando un modelo en tiempo de ejecución que es agnóstico a la aplicación que se ejecuta [18, 19]. Aunque estos enfoques obtienen un rendimiento excelente en la mayoría de los códigos, para problemas irregulares complejos como los que abordamos en esta tesis, una optimización general sin conocimiento de la aplicación se queda corta para aprovechar más eficientemente los recursos de la arquitectura.

Esta tesis está motivada por el hecho de que aún no hay una implementación eficiente para arquitecturas heterogéneas de dos problemas de análisis masivos de datos, reales y complejos, ampliamente utilizados en diversos campos del Big Data: las series temporales y la computación del skyline. Por un lado, para las series temporales nos centramos en el problema de descubrimiento de similitudes/discordancias, tomando como punto de partida el algoritmo que representa el estado del arte: Matrix Profile [20]. Las primeras implementaciones del Matrix Profile [20] realizaban una paralelización trivial del problema, pero con una importante carga computacional por iteración. Las implementaciones más modernas [21, 22] optimizan el cálculo secuencial del Matrix Profile reordenando y evitando operaciones innecesarias, pero a costa de crear un problema irregular. La carga de trabajo de estas implementaciones puede ser modelada y regularizada en tiempo de ejecución para obtener una distribución de carga de trabajo óptima para diferentes aceleradores, como proponemos en el capítulo 3 (para GPU) y el capítulo 4 (FPGA).

Por otro lado, la computación del skyline es una aplicación muy irregular sin posibilidad de modelar con precisión en tiempo de ejecución. El skyline es un problema de optimización ampliamente utilizado para la toma de decisiones multicriterio. En esta tesis nos centramos en el cálculo del skyline para consultas en *streaming*. La carga de trabajo de cada consulta depende en gran medida de la distribución multidimensional de los puntos. Por lo tanto, hasta que no se recorre el espacio de iteraciones completo es imposible modelar con precisión la carga de trabajo del conjunto de datos de la consulta. Este tipo de irregularidades

ofrece la posibilidad de optimización utilizando una partición dinámica de grano fino para mantener la arquitectura heterogénea ocupada mientras se mantiene el equilibrio entre los dispositivos durante toda la ejecución. Como punto de partida en esta tesis, tomamos los algoritmos de computación del skyline que representan el estado del arte para CPU [23] y GPU [24], como detallaremos en el capítulo 5.

En resumen, en esta tesis se proponen modelos, estrategias de programación y soporte en tiempo de ejecución para estas dos aplicaciones con el fin de optimizar el rendimiento y el consumo de energía en diferentes arquitecturas heterogéneas del tipo CPU+GPU y CPU+FPGA.

A.2. Series temporales en procesadores heterogéneos CPU + GPU

Una serie temporal es una colección de observaciones tomadas secuencialmente, como la del electrocardiograma que se muestra en la Figura A.1.

El análisis de series temporales abarca muchos campos, desde computación en la nube [48, 49, 50], forecasting [51, 52, 53, 54], clustering [55, 56], búsqueda de similitudes [57], geología [58], geodesia [59], o economía [60, 61].

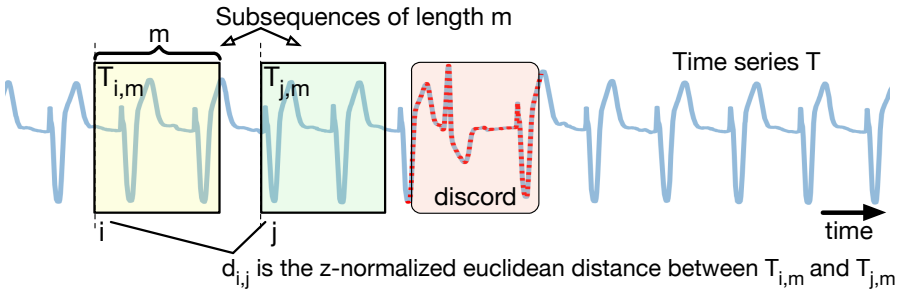


Figura A.1: Serie temporal de electrocardiograma T y dos subsecuencias de las que podemos calcular la distancia $d_{i,j}$. El objetivo es encontrar similitudes/discordancias, como la arritmia ventricular resaltada en el recuadro rojo. La notación se define en la sección 3.1.

En particular, el descubrimiento de similitudes (motivos) o puntos críticos (discordancias) en una serie temporal es relevante para varios de los problemas anteriores. Las similitudes y las discordancias pueden encontrarse mediante enfo-

ques probabilísticos [62, 63], aprendizaje automático [64] o proponiendo modelos espaciotemporales [65].

Una solución recientemente propuesta [66] consiste en calcular primero una puntuación para cada subsecuencia de la serie temporal, lo que da lugar a otra serie temporal denominada Matrix Profile. Mediante una simple inspección del Matrix Profile es sencillo identificar las similitudes y las discordancias centrándose en los valores mínimos y máximos respectivamente. Se han propuesto varios algoritmos para calcular el Matrix Profile: STAMP [66], STOMP [20], SCRIMP [21] o SCAMP [22], que presenta un mayor grado de paralelismo. Asimismo, se han implementado diferentes versiones de SCRIMP para portarlo a multiprocesadores de memoria compartida, sistemas de memoria distribuida [70], plataformas multi-GPU [22] y procesadores Intel Xeon Phi KNL [71]. En el momento en que se desarrolló este trabajo, SCRIMP era el algoritmo del estado del arte para calcular el Matrix Profile.

En este capítulo se propone una implementación heterogénea de CPU + GPU del Matrix Profile utilizando el algoritmo SCRIMP [21] como punto de partida. Prestamos especial atención al problema de desequilibrio de carga que plantea el algoritmo, donde cada iteración paralela tiene una carga de trabajo computacional diferente. Como explicamos más adelante, esto es una consecuencia del patrón computacional que sigue el recorrido en diagonal de una matriz triangular donde cada diagonal tiene una longitud diferente. Para resolver este problema, evaluamos la eficiencia del planificador por defecto de Threading Building Blocks [152], TBB, que se basa en una partición dinámica del espacio de iteración y lo comparamos con una distribución estática basada en una partición que estima con un modelo analítico la carga de trabajo.

Además, contribuimos con una versión heterogénea del Matrix Profile que distribuye dinámicamente el cómputo del Matrix Profile entre los núcleos de la CPU y una GPU. Para ello, partimos de una plantilla paralela desarrollada previamente [117] que implementa un planificador heterogéneo sobre TBB. Este planificador se encarga de balancear la carga entre dispositivos, en nuestro caso la CPU y GPU, y de encontrar la granularidad adecuada para los bloques de trabajo que se procesan en cada uno. pra nuestra implementación heterogénea del Matrix Profile nos encontramos con el reto adicional de qeu además es necesario incoporar un patrón parallel_reduce heterogéneo. Por ello, en este capítulo proponemos algunas estrategias para extender nuestro anterior patrón parallel_for para incorporar también reducciones paralelas heterogéneas. Más concretamente, la memoria virtual compartida, SVM, se aprovecha para minimizar la sobrecarga de comunicación CPU-GPU. Proponemos dos alternativas para el código OpenCL que implementa el cómputo del Matrix Profile en la GPU.

Una precisa pero más lenta que se basa en las operaciones atómicas de OpenCL para garantizar la correcta implementación de las operaciones de reducción. Y otra alternativa imprecisa pero más rápida que evita las operaciones atómicas de OpenCL pero provoca pérdidas de precisión.

Evaluamos experimentalmente tres planificadores heterogéneos (*Static*, *Dynamic* y *LogFit*) [18] que requieren una entrada diferente del usuario. Los que requieren información por parte del programador: *Static* que necesita el porcentaje de trabajo a descargar en la GPU, o *Dynamic* que requiere del tamaño del bloque de iteraciones que debe ser enviado dinámicamente a la GPU. Y otro planificador que no requiere intervención del programador, *LogFit*, que es un plafinicador adaptativo que representa el estado del arte y que calcula automáticamente la granularidad de los bloques de trabajo que se envían de manera dinámica y adaptativa a los distintos dispositivos del sistema. Los tamaños se calculan automáticamente, en tiempo de ejecución, en función del throughput de cada dispositivo a lo largo de la ejecución del espacio de iteraciones, teniendo como objetivo optimizar el throughput de cada dispositivo mientras se minimiza el desbalanceo. Los resultados experimentales muestran que las estrategias de planificación dinámicas siempre mejoran el rendimiento de la partición estática, en particular cuando el número de núcleos aumenta. Aunque el particionamiento estático calcula analíticamente una distribución casi perfecta de las diagonales, no puede hacer frente a las asimetrías en tiempo de ejecución de la carga en los núcleos de la CPU, incluso cuando los experimentos se realizaron en escenarios sin carga (no había otras aplicaciones de usuario ejecutándose en los sistemas).

Nuestro análisis del rendimiento y la eficiencia energética de las implementaciones heterogéneas de CPU+GPU nos ha enseñado que:

- Si la precisión no es un problema, las versiones no atómicas tienen un mejor rendimiento que las atómicas. En este caso, el planificador heterogéneo *Dynamic* puede lograr el tiempo de ejecución más rápido. Sin embargo, el usuario tiene que encontrar el tamaño óptimo del bloque de GPU (requiere de un entrenamiento previo a la ejecución).
- Si la precisión es un requisito, el planificador heterogéneo *LogFit* suele proporcionar la ejecución más rápida. Como ventaja adicional, este planificador busca automáticamente el tamaño óptimo de los bloques GPU sin intervención del usuario.
- Si el consumo de energía es el criterio objetivo, entonces la ejecución sólo en GPU (tanto para las versiones sin atómicos como con atómicos) es la recomendada.

- Nuestra implementación heterogénea nos permite alcanzar un rendimiento casi ideal en procesadores heterogéneos que incorporan una GPU en el chip.

A.3. Series temporales en procesadores heterogeneos CPU + FPGA

El Capítulo 4 propone un novedoso planificador jerárquico llamado *Fastfit*, para equilibrar eficientemente la carga de trabajo en un sistema heterogéneo garantizando un rendimiento casi óptimo. Hemos utilizado *SCAMP* - el nuevo estado del arte para el cálculo del Matrix Profile - para ilustrar su aplicabilidad. *Fastfit* es un planificador a nivel de sistema basado en un modelo analítico del pipeline (IP) que sintetiza un kernel en FPGA. Nuestro modelo nos ayuda a encontrar el tamaño del bloque de trabajo que enviar a la FPGA para garantizar un rendimiento casi óptimo de la FPGA y, a partir de él, el tamaño del bloque que enviar a la CPU para garantizar el equilibrio de carga entre los dispositivos. Además, *Fastfit* incluye un planificador a nivel de dispositivo que proporciona una partición efectiva del bloque de la FPGA en sub-bloques para cada IP de la FPGA, en caso de que el kernel se haya replicado.

A través de una evaluación exhaustiva, validamos la precisión de nuestros modelos y la eficiencia de *Fastfit* para obtener el tamaño del bloque de FPGA casi óptimo, encontrando que nuestro modelo es capaz de encontrar un bloque y predecir entre el 97 %-99 % del mejor rendimiento real medido. También comparamos diferentes estrategias para llevar a cabo la partición a nivel de dispositivo del bloque de la FPGA entre IPs, encontrando que la estrategia *Balanceada*, que es consciente de la geometría triangular del problema, mejora el rendimiento de una estrategia de bloque constante - que divide el tamaño del bloque entre el número de IPs - en un 16,45 %. También proponemos que para mejor explotar la HBM de la que dispone la FPGA con la que trabajamos, podemos utilizar un modelo de uso del ancho de banda de la HBM, modelo que utilizamos para controlar el número de bancos que se asigna a cada IP y de esta manera establecer el número mínimo de bancos activos que garanticen el ancho de banda de la memoria agregada requerido por la aplicación y el tamaño de los datos de entrada, a la vez que se reduce el consumo de energía.

Validamos experimentalmente nuestro planificador en términos de rendimiento y consumo de energía y lo comparamos con otros planificadores heterogéneos anteriores que representan el estado del arte. Demostramos que nuestro nuevo planificador mejora todas las propuestas previas en términos de rendimiento.

Además, *Fastfit* es un 4,68 % mejor que *Logfit* - el estado del arte en planificación adaptativa para dispositivos heterogéneos que no requiere intervención del programador -. La razón de esta mejora es que nuestra nueva propuesta evita la sobrecarga del ajuste logarítmico que realiza *Logfit* durante una fase de entrenamiento, al principio de la ejecución del lazo paralelo. De hecho, *Fastfit* está a sólo un 0,6 % de la ejecución heterogénea ideal - sin la sobrecarga del planificador -. Analizando el consumo de energía, se puede observar que la FPGA presenta la mayor eficiencia energética y el menor consumo de energía. La CPU requiere casi 3 veces más energía para realizar el mismo cálculo. Ahora bien, la energía consumida por los planificadores heterogéneos es aproximadamente proporcional a la carga de trabajo procesada por cada dispositivo (CPU y FPGA). Por ejemplo, *Static* descarga más trabajo a la FPGA (64 % de los elementos) y, en consecuencia, muestra una mejor eficiencia energética que *Dynamic*, *Logfit* y *Fastfit*. En realidad, el consumo de energía y la eficiencia energética en estos tres últimos planificadores son similares. Debido a que *Logfit* es el más lento de los tres últimos planificadores, también consume más energía que *Dynamic* y *Fastfit*.

En resumen, *Fastfit* es el mejor planificador si nuestro objetivo es conseguir el máximo rendimiento. Aunque *Dynamic* también consigue buenos resultados, recordemos que en este caso el programador necesita explorar previamente a la ejecución, y de forma exhaustiva, todos los posibles tamaños de bloque para encontrar el casi óptimo, mientras que en *Fastfit* el mejor tamaño de bloque se descubre automáticamente en tiempo de ejecución. Sin embargo, si nuestro objetivo es minimizar el consumo de energía, descargar toda la carga de trabajo a la FPGA es la mejor opción. Utilizar sólo la FPGA reduce en un 75,78 % el consumo de energía y mejora en un 43,11 % la eficiencia energética (Elements/J) en comparación con *Fastfit*.

A.4. Cálculo del Skyline en procesadores heterogéneous CPU + GPU

El skyline, introducido inicialmente en [86] es un problema de optimización ampliamente utilizado para la toma de decisiones multicriterio. Permite minimizar un conjunto de datos de N dimensiones en el subconjunto más pequeño, normalmente utilizando como métrica de reducción el valor *mínimo* para cada dimensión.

Se ha aplicado en muy diversos contextos tales como preservar la privacidad de los datos en múltiples dominios [87], el procesamiento del skyline sobre los

datos cifrados en las bases de datos en la nube [88], en optimización de "Quality-of-Services" de los procesos de grandes servicios, usando el skyline como método para descubrir servicios [89, 90], o en servicios de drones para la entrega en condiciones climáticas dinámicas [91]. También se ha aplicado en el contexto de aprendizaje por refuerzo para mejorar la planificación adaptativa en los servicios de computación en la nube [92], o sobre datos encriptados de múltiples fuentes para una fusión de datos eficiente que preserve la privacidad [93], o la minería evolutiva de la agrupación de grafos [94].

Punto	x	y
A	3	3
B	1	3
C	2	1

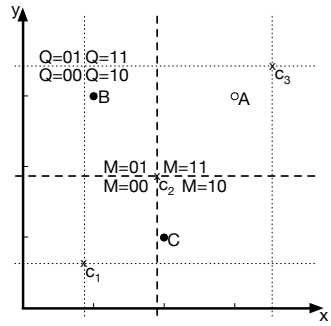


Figura A.2: Dataset y ejemplo de skyline. Los puntos del skyline son B y C.

Figura A.2 muestra un ejemplo sencillo de un conjunto de datos y su correspondiente skyline. El punto C tiene valores más bajos en sus dos dimensiones con respecto al punto A, por lo que el punto A es eliminado del skyline por el punto C. Por lo tanto, el skyline es el subconjunto de puntos que no son eliminados (no tienen valores inferiores en todas sus dimensiones) por ningún otro punto del conjunto de datos. El punto B tiene un valor menor en x que el punto C, pero mayor en y, por lo que ninguno puede eliminar al otro y ambos, B y C, acaban en el conjunto del skyline. Para grandes conjuntos de datos con puntos que tienen varias dimensiones, el cálculo del skyline se convierte en una tarea computacionalmente costosa.

Para aumentar el rendimiento del skyline, es fundamental evitar la comparación de todos los puntos. Para ello, se suelen adoptar dos enfoques: (1) un algoritmo basado en la ordenación; y (2) un algoritmo basado en la partición. El algoritmo secuencial del estado del arte utiliza un enfoque basado en la partición recursiva, que no se adapta bien a las arquitecturas heterogéneas, como por ejemplo la GPU. Los algoritmos multinúcleo actuales explotan estrategias de ordenación [23] que explotan más paralelismo a costa de aumentar la carga

computacional del problema. Los algoritmos actuales para la arquitectura GPU ofrecen una solución intermedia, siguiendo una estrategia de partición no recursiva, que aumenta considerablemente el coste computacional del problema, pero también libera un paralelismo masivo (clave en las GPUs) [24]. El problema del cálculo del skyline sobre un *stream* (flujo) de datos plantea retos adicionales. El cálculo del skyline requiere el conocimiento de todo el conjunto de datos antes de iniciar el cómputo, lo que supone un desafío cuando los datos llegan en *stream*. Los enfoques secuenciales o paralelos actuales [110, 108, 109, 171] calculan el skyline utilizando estrategias de ventanas deslizantes sobre los puntos, produciendo un flujo de actualizaciones del skyline con la llegada de nuevos puntos.

El Capítulo 5 aborda el problema del cálculo del skyline sobre un *stream* de datos independientes en una arquitectura heterogénea compuesta por una CPU multinúcleo y una GPU integrada. En nuestro trabajo cada entrada no es un punto, sino un conjunto de datos completo. Así, obtenemos un flujo de salida de un skyline por cada entrada recibida. Para ello, proponemos una solución basada en grafos heterogéneos, denominada SkyFlow, para planificar eficientemente la carga de trabajo entre los dispositivos. Proponemos dos enfoques que se adaptan a diferentes escenarios de *streaming*: Coarse-grained (SkyFlow-CG) y Fine-grained (SkyFlow-FG).

SkyFlow-CG calcula simultáneamente un skyline por dispositivo, utilizando un enfoque híbrido: cada dispositivo ejecuta el algoritmo que mejor se adapta a las características específicas de la arquitectura correspondiente. Así, la CPU ejecuta el algoritmo del estado del arte en CPU, *Hybrid* - basado en una implementación OpenMP -, mientras que la GPU ejecuta el algoritmo del estado del arte en GOU, *SkyAlign* - basado en una implementación SYCL, novedosa en este trabajo - . Aunque ambos algoritmos aprovechan la eficiencia del trabajo reduciendo el número de pruebas de dominancia necesarias durante el cálculo del skyline, para los conjuntos de datos reales evaluados en este trabajo hemos encontrado que algunos de ellos rinden mejor bajo OpenMP-CPU, mientras que otros bajo SYCL-GPU. Esto se debe a la naturaleza altamente irregular de este operador, cuya carga computacional depende de la configuración del conjunto de datos de llegada (distribución de puntos en el espacio, tamaño, número de dimensiones), y del algoritmo y la arquitectura destino.

Para el enfoque SkyFlow-CG consideramos dos estrategias de planificación: la conservación del trabajo (SkyFlow-CG WC) y el tiempo de finalización más temprano heterogéneo (SkyFlow-CG HEFT). Mientras que WC tiene como objetivo mantener todos los dispositivos ocupados enviando a la cola del dispositivo más corta cualquier conjunto de datos que llegue, la estrategia HEFT intenta optimizar el rendimiento de cada dispositivo colocando el conjunto de datos entrante

en la cola del dispositivo en la que terminará antes, independientemente de si un dispositivo está inactivo. HEFT requiere estimar en tiempo de ejecución del cálculo del skyline sobre el conjunto de datos que llega a cada dispositivo. Para ello, en este trabajo introducimos un novedoso modelo que se basa en un muestreo inicial de algunos puntos del conjunto de datos, ejecutado bajo SYCL-GPU y OpenMP-CPU. A través de una evaluación exhaustiva hemos encontrado que la inexactitud en la que incurre nuestro modelo está dentro del $\pm$ 10 % de los tiempos reales de cálculo del skyline, que siempre es menor que la diferencia de tiempo entre algoritmos-dispositivos. En consecuencia, nuestro modelo siempre selecciona el dispositivo apropiado. En cualquier caso, en nuestra evaluación de las estrategias de programación, HEFT siempre supera a WC en todos los escenarios de streaming y conjuntos de datos. En particular, la estrategia HEFT supera a WC hasta 3,32 veces en escenarios que contienen una mezcla aleatoria de conjuntos de datos que pueden ser óptimos para cada algoritmo-dispositivo. Además, cuando comparamos el rendimiento de nuestra heurística HEFT con una estrategia *Oracle* que calcula, previamente a la ejecución, el dispositivo óptimo para cada conjunto de datos, comprobamos que HEFT sólo degrada el rendimiento de *Oracle* en un 6 % como máximo. Esto corrobora que nuestra heurística basada en el modelo introduce una baja sobrecarga, al tiempo que garantiza una planificación casi óptima.

En segundo lugar, el enfoque SkyFlow-FG particiona dinámicamente la carga de trabajo de cada conjunto de datos entre SYCL-GPU y SYCL-CPU. Mediante una cuidadosa selección del tamaño (y número) de los trozos de datos enviados a cada dispositivo, hemos comprobado que esta estrategia de partición de grano fino es beneficiosa en un escenario de *streaming* en el que la mayoría de los conjuntos de datos se ejecutan más rápidamente en GPU. En este escenario, SkyFlow-FG mejora hasta en 2 veces el rendimiento cuando se compara con una ejecución en solo GPU, y supera a SkyFlow-CG HEFT en 1,5 veces.

A.5. Conclusiones

Este capítulo resume los resultados de esta tesis. Proponemos un conjunto de estrategias para optimizar en tiempo de ejecución dos aplicaciones irregulares de análisis masivo de datos en arquitecturas heterogéneas: el Matrix Profile y el skyline en streaming. Nuestro objetivo es explorar modelos de programación que nos permitan minimizar el esfuerzo de programación del usuario mientras se obtiene el máximo rendimiento de estas aplicaciones cuando se portan a arquitecturas heterogéneos que inclyen aceleradores.

El Capítulo 3 explora diferentes alternativas para mapear de forma óptima el algoritmo Matrix Profile en procesadores heterogéneos con varios núcleos de CPU y una GPU integrada. Para aprovechar mejor esta plataforma heterogénea podemos descargar parte de la carga computacional en la GPU. Con este objetivo, proponemos HetMP, basado en TBB, que es una extensión de una librería previamente diseñada para facilitar la implementación de aplicaciones en arquitecturas heterogéneas, incluyendo ahora nuevas funcionalidades de sincronización y así como de operaciones de reducción sobre estas arquitecturas. Proponemos y analizamos dos versiones del *kernel* de Matrix Profile, *NonAtomic* (más rápida pero insegura) y *Atomic* (más lenta pero precisa), concluyendo que la versión *Atomic* sólo incurre en un 8,9 % de degradación del rendimiento respecto a la *NonAtomic* para la ejecución sólo en la GPU. Se han evaluado tres alternativas para distribuir la carga de trabajo entre la CPU y la GPU (*Static*, *Dynamic* y *LogFit*), encontrando que la más productiva es *LogFit* cuando se requiere una solución exacta. Sin embargo, si la magnitud a minimizar es el consumo de energía, es aconsejable descargar todo el cómputo en la GPU. En futuros trabajos ampliaremos el planificador heterogéneo para que pueda ser instruido en la optimización de otras métricas, aparte de maximizar el rendimiento: por ejemplo el minimizar el consumo de energía, o bien maximizar una métrica que represente un compromiso entre rendimiento y energía.

En el Capítulo 4 estudiamos el problema de ejecutar eficientemente el algoritmo que representa el estado del arte actual para el cómputo de series temporales -*SCAMP*-, que en este caso trasladamos a una plataforma heterogénea compuesta por CPU + FPGA de alto rendimiento con bancos de memoria HBM[1] integrada. La geometría del algoritmo (una matriz triangular) y las capacidades de la FPGA plantean dos retos. En primer lugar, se pueden instanciar varias IPs replicadas en la estructura de la FPGA, por lo que el reparto equilibrado de la carga es un problema no sólo a nivel del sistema (CPU+FPGA), sino también a nivel de dispositivo (entre las IPs de la FPGA, donde cada IP representa la síntesis de un kernel Matrix Profile). Y en segundo lugar, los datos a los que accede cada una de estas IPs deben ser cuidadosamente colocados entre los bancos HBM para explotar eficientemente el ancho de banda de la memoria que ofrecen los bancos, al tiempo que se selecciona el número mínimo de bancos activos que optimiza el consumo de energía.

Para hacer frente al primer reto proponemos un novedoso planificador jerárquico llamado *Fastfit*, para equilibrar eficientemente la carga de trabajo en el sistema heterogéneo, garantizando al mismo tiempo un rendimiento casi óptimo. Nuestro planificador consta de un soporte en tiempo de ejecución de dos niveles:

[1] HBM=*High Bandwidth Memory*

1) el nivel de sistema, que aprovecha un modelo analítico de la FPGA, para encontrar el tamaño casi óptimo de los bloques que se asignan a la FPGA y que han de garantizar un rendimiento óptimo de la FPGA; y 2) el nivel de dispositivo, que tiene en cuenta la geometría del problema y que es responsable de la partición efectiva del bloque asignado a la FPGA en sub-bloques que se envían a cada IP de la FPGA. Por otro lado, para hacer frente al segundo reto, proponemos una metodología basada en un modelo de uso del ancho de banda de la¡os bancos de HBM que nos permite establecer el número mínimo de bancos activos que garanticen el máximo ancho de banda de la memoria agregada para un número determinado de IPs. A través de una evaluación exhaustiva, validamos la precisión de nuestros modelos, la eficiencia de nuestras estrategias de partición a nivel de sistema y a nivel de los dispositivos, así como el rendimiento y la eficiencia energética de nuestro planificador heterogéneo *Fastfit*, descubriendo que nuestra propuesta supera a los planificadores anteriores que representan del estado del arte, alcanzando hasta el 99,4 % del rendimiento ideal.

El Capítulo 5 aborda el cómputo del operador skyline sobre un *stream* (flujo) de consultas de datos independientes en una arquitectura heterogénea compuesta por una CPU multinúcleo y una GPU integrada. El skyline es un operador de optimización ampliamente utilizado para la toma de decisiones multicriterio. Permite minimizar un conjunto de datos de N dimensiones en su subconjunto más pequeño. En este trabajo presentamos SkyFlow, la primera solución en tiempo de ejecución que se basa en grafos heterogéneos CPU+GPU que permite el cómputo del skyline sobre un *stream* de conjuntos de datos. Se han propuesto dos enfoques de flujo de datos, Coarse-grained y Fine-grained, para diferentes escenarios de streaming. El objetivo de Coarse-grained es mantener el cálculo de dos skylines en paralelo utilizando una solución híbrida con dos algoritmos de skyline diferentes y óptimizados, uno para la CPU y otro para la GPU. También proponemos un modelo para estimar en tiempo de ejecución del skyline sobre cualquier conjunto de datos que llegue, estimación utilizada por una heurística para planificar el conjunto de datos en la cola del dispositivo en el que terminará antes, tratando de optimizar de esta manera el rendimiento global del sistema. Por otro lado, Fine-grained divide el cómputo de un conjunto de datos entre la CPU y la GPU. Nuestros resultados experimentales muestran que, en nuestros escenarios de streaming y para los conjuntos de datos evaluados, nuestros enfoques heterogéneos de CPU+GPU siempre superan a las implementaciones anteriores que representan el estado del arte en un multicore CPU y en una GPU, hasta en 6,86 veces y 5,19 veces, respectivamente, y se sitúan sólo por debajo del 6 % del rendimiento máximo ideal.

A.6. Limitaciones

Aunque esta tesis aporta varias contribuciones, también tiene algunas limitaciones. La primera limitación está relacionada con la portabilidad del código. Para poder usar los planificadores propuestos en los capítulos 3 y 4, se requeriere que el usuario proporcione dos versiones del kernel, una para los núcleos de la CPU y otra para la GPU/FPGA. Sin embargo, sería beneficioso codificar una única versión del kernel para que este se compile y ejecute en cada dispositivo del sistema. En este sentido, estamos trabajando en portar el código OpenCL existente a DPC++, utilizando para ello la capa de abstracción de alto nivel que proporciona SYCL, puesto que soporta todas las características de OpenCL. DPC++ permite la comodidad, la productividad y la flexibilidad del paradigma único código fuente expresado en el estándar C++. SYCL permite que el código del kernel quede incrustado en el código del host usando el mismo lenguaje de programación, por lo que los programadores ganan en simplicidad al codificar la aplicación y los compiladores ganan en capacidad de analizar y optimizar todo el programa, independientemente del dispositivo en el que se vaya a ejecutar el código. DPC++ logra esto a través de múltiples pases de compiladores sobre un único fuente (SMCP). Con SMCP, un único archivo fuente es analizado por diferentes compiladores para diferentes dispositivos destino, generando diferentes binarios (uno por dispositivo). En muchos casos, esos binarios se combinan en un único ejecutable. De este modo, podremos dirigirnos a diferentes dispositivos con el mismo código fuente, como ilustramos por ejemplo en el trabajo realizado en el capítulo 5.

Otra limitación de esta tesis es que nuestras estrategias de planificación consideran que nuestra aplicación es la única que se ejecuta en el sistema. Sin embargo, como suele ocurrir en las arquitecturas heterogéneas, puede que varias aplicaciones estén ejecutándose al mismo tiempo. Esto supone que nuestros modelos analíticos y planificadores deben de ajustarse para evitar sobreestimar las estimaciones de rendimiento, y el que la carga de trabajo entre los dispositivos no esté correctamente balanceada. Para ello, es necesario seguir investigando en escenarios multi-aplicación, en los que nuestros enfoques dinámicos y adaptativos podrían ajustarse para incluir los efectos de otras aplicaciones cuando compiten por los recursos.

A.7. Trabajos futuros

Para concluir esta tesis, queremos proponer futuras líneas de investigación relacionadas con nuestro trabajo.

- Como se ha mencionado en el Capítulo 3, nuestro planificador heterogéneo está enfocado siempre a buscar el máximo rendimiento. Podríamos ampliar nuestro planificador heterogéneo para que también pueda optimizar otras métricas, como el consumo de energía u otra métrica derivada que considere el rendimiento y la energía al mismo tiempo.

- En cuanto al Capítulo 4, nuestro acelerador, la FPGA que integra bancos de memoria HBM, utiliza aritmética de punto flotante para realizar las operaciones. Tenemos previsto explorar el uso de aritmética de punto fijo para optimizar la síntesis de los IPs de la FPGA. La aritmética de punto fijo no sólo permite realizar operaciones más rápidas que las de punto flotante, sino que también necesita menos hardware para ser implementada. De este modo, el uso de esta aritmética mejoraría el rendimiento de la implementación de los IPs en la FPGA y su consumo de energía. Recientemente, el fabricante ha lanzado un BSP para soportar oneAPI en la FPGA HBM. Por lo tanto, una migración de código OpenCL a DPC++ puede abordarse, lo que permitiría la explotación de los bancos de memoria HBM mediante SYCL. Además, nuestro planificador jerárquico, Fastfit, podría extenderse para incorporar otros aceleradores disponibles en el sistema, como una GPU integrada/discreta. Para ello, el planificador debería incluir estos nuevos dispositivos en la estrategia de partición/planificación. Por último, el planificador también podría ampliarse para que incorpore otras métricas, como minimizar el consumo de energía u optimizar una métrica derivada de relación rendimiento/energía, que son de especial interés cuando se utiliza un dispositivo de muy bajo consumo, como una FPGA.

- Por último, en el Capítulo 5, tenemos previsto explorar el uso de la FPGA como dispositivo acelerador adicional para computar el operador skyline sobre un stream de conjuntos de datos. Una FPGA con soporte para bancos HBM podría mejorar el rendimiento de las implementaciones de FPGA existentes para los cálculos del skyline, una aproximación que no se ha explorado. La adición de este nuevo acelerador podría dar lugar a combinaciones de ejecuciones Coarse-grained (SkyFlow-CG) con ejecuciones Fine-grained (SkyFlow-FG). Por ejemplo, podrían ejecutarse en paralelo dos skylines, uno de ellos en la FPGA aprovechando al máximo las capacidades de la

HBM - siguiendo el enfoque de SkyFlow-CG -, y el segundo en la CPU - siguiendo el enfoque SkyFlow-FG -.

Bibliography

[1] Scott E Thompson and Srivatsan Parthasarathy. Moore's law: the future of si microelectronics. *Materials today*, 9(6):20–25, 2006. (Cited on pages 2 and 156)

[2] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008. https://doi.org/10.1109/MC.2008.209. (Cited on pages 2 and 156)

[3] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, and Amit Bawaskar. GPGPU processing in CUDA architecture. *CoRR*, abs/1202.4347, 2012. URL: http://arxiv.org/abs/1202.4347, https://arxiv.org/abs/1202.4347. (Cited on page 2)

[4] Andrew S Glassner. *An introduction to ray tracing*. Morgan Kaufmann, 1989. (Cited on page 2)

[5] Ian Kuon, Russell Tessier, and Jonathan Rose. *FPGA architecture: Survey and challenges*. Now Publishers Inc, 2008. (Cited on page 2)

[6] Gene Frantz. Digital signal processor trends. *IEEE micro*, 20(6):52–59, 2000. (Cited on page 3)

[7] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017. (Cited on page 3)

[8] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010. (Cited on page 4)

[9] David R Kaeli, Perhaad Mistry, Dana Schaa, and Dong Ping Zhang. *Heterogeneous computing with OpenCL 2.0*. Morgan Kaufmann, 2015. (Cited on page 4)

[10] Ronan Keryell, Ruyman Reyes, and Lee Howes. Khronos sycl for opencl: a tutorial. In *Proceedings of the 3rd International Workshop on OpenCL*, pages 1–1, 2015. (Cited on page 4)

[11] James Reinders, Michael Voss, Pablo Reble, Rafael Asenjo-Plaza, et al. C++ for heterogeneous programming: oneapi (dpc++ and onetbb), 2020. (Cited on page 4)

[12] Hervé Guihot. Renderscript. In *Pro Android Apps Performance Optimization*, pages 231–263. Springer, 2012. (Cited on page 4)

[13] Mare-sdk. https://developer.qualcomm.com/software/mare-sdk, 2021. Accessed: 2021-11-25. (Cited on page 4)

[14] Alexander Collins, Christian Fensch, Hugh Leather, and Murray Cole. Masif: Machine learning guided auto-tuning of parallel skeletons. In *20th Annual International Conference on High Performance Computing*, pages 186–195. IEEE, 2013. (Cited on pages 5 and 157)

[15] Dominik Grewe, Zheng Wang, and Michael FP O'Boyle. Opencl task partitioning in the presence of gpu contention. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 87–101. Springer, 2013. (Cited on pages 5 and 157)

[16] Dominik Grewe and Michael FP O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *International conference on compiler construction*, pages 286–305. Springer, 2011. (Cited on pages 5 and 157)

[17] CK Luk, S Hong, and KH Qilin. Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*, pages 45–55, 2010. (Cited on pages 5 and 157)

[18] Angeles Navarro, Francisco Corbera, Andres Rodriguez, Antonio Vilches, and Rafael Asenjo. Heterogeneous parallel_for template for cpu–gpu chips. *International Journal of Parallel Programming*, 47(2):213–233, 2019. (Cited on pages 5, 8, 157 and 160)

[19] Antonio Vilches, Angeles Navarro, Rafael Asenjo, Francisco Corbera, Ruben Gran, and Maria J Garzaran. Mapping streaming applications on commodity multi-cpu and gpu on-chip processors. *IEEE Transactions on Parallel and Distributed Systems*, 27(4):1099–1115, 2015. (Cited on pages 5 and 157)

[20] Y. Zhu, Z. Zimmerman, N. S. Senobari, C. M. Yeh, G. Funning, A. Mueen, P. Brisk, and E. Keogh. Matrix profile ii: Exploiting a novel algorithm and GPUs to break the one hundred million barrier for time series motifs and joins. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 739–748, Dec 2016. https://doi.org/10.1109/ICDM.2016.0085. (Cited on pages 5, 30, 39, 40, 69, 157 and 159)

[21] Yan Zhu, Chin-Chia Michael Yeh, Zachary F Zimmerman, Kaveh Kamgar, and E. Keogh. Matrix Profile XI: SCRIMP++: Time series motif discovery at interactive speeds. *2018 IEEE International Conference on Data Mining (ICDM)*, pages 837–846, 2018. (Cited on pages 5, 8, 30, 39, 52, 69, 75, 77, 157 and 159)

[22] Yan Zhu, Zachary Zimmerman, Nader Shakibay Senobari, Chin-Chia Michael Yeh, Gareth Funning, Abdullah Mueen, Philip Brisk, and Eamonn Keogh. Exploiting a novel algorithm and GPUs to break the ten quadrillion pairwise comparisons barrier for time series motifs and joins. *Knowledge and Information Systems*, 54(1):203–236, Jan 2018. URL: https://doi.org/10.1007/s10115-017-1138-x, https://doi.org/10.1007/s10115-017-1138-x. (Cited on pages 5, 8, 30, 39, 49, 52, 77, 157 and 159)

[23] Sean Chester, Darius Šidlauskas, Ira Assent, and Kenneth S Bøgh. Scalable parallelization of skyline computation for multi-core processors. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1083–1094. IEEE, 2015. (Cited on pages 6, 33, 109, 113, 115, 158 and 163)

[24] Kenneth S Bøgh, Sean Chester, and Ira Assent. Skyalign: a portable, work-efficient skyline algorithm for multicore and gpu architectures. *The VLDB Journal*, 25(6):817–841, 2016. (Cited on pages 6, 33, 110, 113, 118, 120, 122, 158 and 164)

[25] Angeles Navarro, Antonio Vilches, Francisco Corbera, and Rafael Asenjo. Strategies for maximizing utilization on multi-cpu and multi-gpu heterogeneous architectures. *The Journal of Supercomputing*, 70(2):756–771, 2014. (Cited on page 8)

[26] José Carlos Romero, Maria Angeles Gonzalez-Navarro, Andres Rodriguez-Moreno, Rafael Asenjo-Plaza, and Murray Cole. Time series collaborative execution on cpu+ gpu. *HPC-Europa3 Transnational Access Meeting (TAM)*, 2019. (Cited on page 9)

[27] José Carlos Romero, Maria Angeles Gonzalez-Navarro, Andres Rodriguez-Moreno, Rafael Asenjo-Plaza, and Murray Cole. Time series heterogeneous co-execution on cpu+ gpu. *International Conference on Computational and Mathematical Methods in Science and Engineering*, 2019. (Cited on page 9)

[28] José Carlos Romero, Felipe Muñoz, Antonio Vilches, Maria Angeles Gonzalez-Navarro, Andres Rodriguez-Moreno, and Rafael Asenjo-Plaza. Skyflow: Heterogeneous streaming for skyline computation using flowgraph and oneapi. *VI Congreso Nacional de Informática (CEDI)*, 2019. (Cited on page 9)

[29] Jose C. Romero, Antonio Vilches, Andrés Rodríguez, Angeles Navarro, and Rafael Asenjo. Scrimpco: scalable matrix profile on commodity heterogeneous processors. *The Journal of Supercomputing*, 2020. URL: https://doi.org/10.1007/s11227-020-03199-w, https://doi.org/10.1007/s11227-020-03199-w. (Cited on page 9)

[30] Jose Carlos Romero, Angeles Navarro, Antonio Vilches, Andrés Rodríguez, Francisco Corbera, and Rafael Asenjo. Efficient heterogeneous matrix profile on a cpu+ high performance fpga with integrated hbm. *Future Generation Computer Systems*, 125:10–23, 2021. (Cited on page 9)

[31] Sparsh Mittal and Jeffrey S Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):1–35, 2015. (Cited on page 11)

[32] Top 500 project. https://www.top500.org. Accessed: 2021-12-02. (Cited on page 11)

[33] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965. (Cited on page 12)

[34] Karl Rupp. microprocessor-trend-data. https://github.com/karlrupp/microprocessor-trend-data, 2020. (Cited on pages 12 and 13)

[35] Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017. (Cited on page 12)

[36] Intel. The architecture of intel processor graphics gen11. https://www.intel.com/content/dam/develop/external/us/en/documents/the-architecture-of-intel-processor-graphics-gen11-r1new.pdf, 2019. (Cited on pages 14 and 15)

[37] Intel Corp. Intel® oneapi dpc++ fpga optimization guide. https://www.intel.com/content/www/us/en/develop/documentation/oneapi-fpga-optimization-guide/top.html, 2021. (Cited on pages 16 and 17)

[38] *The Architecture of Intel Stratix10MX with HBM2.* URL: https://www.bittware.com/es/resources/hbm2-2d-fft-oneapi/. (Cited on page 17)

[39] *Triad Benchmark STREAM* . URL: https://www.cs.virginia.edu/stream/ref.html. (Cited on pages 18 and 28)

[40] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming.* Addison-Wesley, Upper Saddle River, NJ, 2011. 2010017618. (Cited on page 19)

[41] *OpenCL Overview.* URL: https://www.khronos.org/opencl/. (Cited on page 20)

[42] *OpenCL Guide.* URL: https://github.com/KhronosGroup/OpenCL-Guide. (Cited on pages 22 and 23)

[43] The Khronos SYCL Working Group. *SYCL 2020 Specification (revision 3)*, 2021. (Cited on pages 23, 24, 34, 35 and 110)

[44] *SYCL Overview* . URL: https://www.khronos.org/sycl/. (Cited on page 24)

[45] Intel Corporation. oneAPI Specification 1.0 Rev. 2. https://spec.oneapi.io/versions/1.0-rev-2/, 2021. Accessed: 2021-10-05. (Cited on pages 24, 34, 110, 111 and 123)

[46] James Reinders, Ben Ashbaugh, James Broadman, Michael Kinsner, John Pennycook, and Xinmin Tian. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL.* Apress, 2021. (Cited on pages 24, 110 and 128)

[47] Michael Voss, Rafael Asenjo, and James Reinders. *Pro TBB: C++ Parallel Programming with Threading Building Blocks.* Apress, 2019. (Cited on pages 26, 27, 28, 123 and 125)

[48] Cédric St-Onge, Nadjia Kara, Omar Abdel Wahab, Claes Edstrom, and Yves Lemieux. Detection of time series patterns and periodicity of cloud computing workloads. *Future Generation Computer Systems*, 109:249 – 261, 2020. URL: http://www.sciencedirect.com/science/article/pii/S0167739X19312440, https://doi.org/https://doi.org/10.1016/j.future.2020.03.059. (Cited on pages 30, 69 and 158)

[49] Walayat Hussain, Farookh Khadeer Hussain, Morteza Saberi, Omar Khadeer Hussain, and Elizabeth Chang. Comparing time series with machine learning-based prediction approaches for violation management in cloud slas. *Future Generation Computer Systems*, 89:464 – 477, 2018. URL: http://www.sciencedirect.com/science/article/pii/S0167739X17330406, https://doi.org/https://doi.org/10.1016/j.future.2018.06.041. (Cited on pages 30 and 158)

[50] Hua Ma, Haibin Zhu, Zhigang Hu, Wensheng Tang, and Pingping Dong. Multi-valued collaborative qos prediction for cloud service via time series analysis. *Future Generation Computer Systems*, 68:275 – 288, 2017. URL: http://www.sciencedirect.com/science/article/pii/S0167739X16303971, https://doi.org/https://doi.org/10.1016/j.future.2016.10.012. (Cited on pages 30 and 158)

[51] Spyros Makridakis, Evangelos Spiliotis, and Vassilios Assimakopoulos. The m4 competition: 100,000 time series and 61 forecasting methods. *International Journal of Forecasting*, 2019. URL: http://www.sciencedirect.com/science/article/pii/S0169207019301128, https://doi.org/https://doi.org/10.1016/j.ijforecast.2019.04.014. (Cited on pages 30 and 158)

[52] Yulai Zhang, Yuchao Wang, and Guiming Luo. A new optimization algorithm for non-stationary time series prediction based on recurrent neural networks. *Future Generation Computer Systems*, 102:738 – 745, 2020. URL: http://www.sciencedirect.com/science/article/pii/S0167739X18332540, https://doi.org/https://doi.org/10.1016/j.future.2019.09.018. (Cited on pages 30, 69 and 158)

[53] Mu-Yen Chen. A high-order fuzzy time series forecasting model for internet stock trading. *Future Generation Computer Systems*, 37:461 – 467, 2014. Special Section: Innovative Methods and Algorithms for Advanced Data-Intensive Computing Special Section: Semantics, Intelligent processing and services for big data Special Section: Advances in Data-Intensive Modelling and Simulation Special Section: Hybrid Intelligence for Growing Internet and its Applications. URL: http://www.sciencedirect.com/science/article/

pii/S0167739X13002045, https://doi.org/https://doi.org/10.1016/j.future.2013.09.025. (Cited on pages 30 and 158)

[54] Kasun Bandara, Christoph Bergmeir, and Slawek Smyl. Forecasting across time series databases using recurrent neural networks on groups of similar series: A clustering approach. *Expert Systems with Applications*, 140:112896, 2020. URL: http://www.sciencedirect.com/science/article/pii/S0957417419306128, https://doi.org/https://doi.org/10.1016/j.eswa.2019.112896. (Cited on pages 30 and 158)

[55] John Paparrizos and Luis Gravano. k-shape: Efficient and accurate clustering of time series. *SIGMOD Rec.*, 45(1):69–76, 2016. URL: http://doi.acm.org/10.1145/2949741.2949758, https://doi.org/10.1145/2949741.2949758. (Cited on pages 30 and 158)

[56] Aladin Crnkić, Igor Ivanović, Vladimir Jaćimović, and Nevena Mijajlović. Swarms on the 3-sphere for online clustering of multivariate time series and data streams. *Future Generation Computer Systems*, 112:11 – 17, 2020. URL: http://www.sciencedirect.com/science/article/pii/S0167739X19310726, https://doi.org/https://doi.org/10.1016/j.future.2020.05.018. (Cited on pages 30 and 158)

[57] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. The lernaean hydra of data series similarity search: An experimental evaluation of the state of the art. *PVLDB*, 12(2):112–127, 2018. URL: http://www.vldb.org/pvldb/vol12/p112-echihabi.pdf. (Cited on pages 30 and 158)

[58] Roberto Tomás, José Luis Pastor Navarro, Marta Béjar Pizarro, Roberta Bonì, Pablo Ezquerro Martín, José Antonio Fernández-Merodo, Carolina Guardiola-Albert, Gerardo Herrera García, Claudia Meisina, Pietro Teatini, Francesco Zucca, Claudia Zoccarato, and Andrea Franceschini. Wavelet analysis of land subsidence time-series: Madrid tertiary aquifer case study, 2020-04-22. (Cited on pages 30, 69 and 158)

[59] Machiel S. Bos, Jean-Philippe Montillet, Simon D. P. Williams, and Rui M. S. Fernandes. *Introduction to Geodetic Time Series Analysis*, chapter 2, pages 29–52. Springer International Publishing, Cham, 2020. URL: https://doi.org/10.1007/978-3-030-21718-1_2, https://doi.org/10.1007/978-3-030-21718-1_2. (Cited on pages 30 and 158)

[60] Dashan Huang, Jiangyuan Li, Liyao Wang, and Guofu Zhou. Time series momentum: Is it there? *Journal of Financial Economics*,

135(3):774 – 794, 2020. URL: http://www.sciencedirect.com/science/article/pii/S0304405X19301953, https://doi.org/https://doi.org/10.1016/j.jfineco.2019.08.004. (Cited on pages 30 and 158)

[61] Eugene F Fama and Kenneth R French. Comparing Cross-Section and Time-Series Factor Models. *The Review of Financial Studies*, 33(5):1891–1926, 08 2019. URL: https://doi.org/10.1093/rfs/hhz089, https://arxiv.org/abs/https://academic.oup.com/rfs/article-pdf/33/5/1891/33209809/hhz089.pdf, https://doi.org/10.1093/rfs/hhz089. (Cited on pages 30 and 158)

[62] Sahar Torkamani and Volker Lohweg. Survey on time series motif discovery. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(2):e1199, 2017. (Cited on pages 30 and 159)

[63] Sahar Torkamani and Volker Lohweg. Survey on time series motif discovery. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(2), 2017. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/widm.1199, https://doi.org/10.1002/widm.1199. (Cited on pages 30 and 159)

[64] Nesreen K. Ahmed, Amir F. Atiya, Neamat El Gayar, and Hisham El-Shishiny. An empirical comparison of machine learning models for time series forecasting. *Econometric Reviews*, 29(5-6):594–621, 2010. URL: https://doi.org/10.1080/07474938.2010.481556, https://arxiv.org/abs/https://doi.org/10.1080/07474938.2010.481556, https://doi.org/10.1080/07474938.2010.481556. (Cited on pages 30 and 159)

[65] Amy McGovern, Derek H. Rosendahl, Rodger A. Brown, and Kelvin K. Droegemeier. Identifying predictive multi-dimensional time series motifs: an application to severe weather prediction. *Data Mining and Knowledge Discovery*, 22(1):232–258, Jan 2011. URL: https://doi.org/10.1007/s10618-010-0193-7, https://doi.org/10.1007/s10618-010-0193-7. (Cited on pages 30 and 159)

[66] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Diego Furtado Silva, Abdullah Mueen, and Eamonn Keogh. Matrix profile i: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets. In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*, pages 1317–1322. IEEE, 2016. (Cited on pages 30, 39, 69, 71 and 159)

[67] Zachary Zimmerman, Kaveh Kamgar, Nader Shakibay Senobari, Brian Crites, Gareth Funning, Philip Brisk, and Eamonn Keogh. Matrix profile xiv: scaling time series motif discovery with gpus to break a quintillion

pairwise comparisons a day and beyond. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 74–86, 2019. (Cited on pages 30, 69, 71 and 75)

[68] Eamonn Keogh. MASS algorithm: Mueen's Algorithm for Similarity Search. https://www.cs.unm.edu/~mueen/FastestSimilaritySearch.html. Accessed: 2019-04-23. (Cited on page 30)

[69] Amir Raoofy, Roman Karlstetter, Dai Yang, Carsten Trinitis, and Martin Schulz. Time series mining at petascale performance. In Ponnuswamy Sadayappan, Bradford L. Chamberlain, Guido Juckeland, and Hatem Ltaief, editors, *High Performance Computing*, pages 104–123, Cham, 2020. Springer International Publishing. (Cited on page 30)

[70] Gabriel Pfeilschifter. Time Series Analysis with Matrix Profile on HPC Systems. Master's thesis, Department of Informatics, Technical University of Munich, Germany, 2019. (Cited on pages 30, 39 and 159)

[71] Ivan Fernandez, Alejandro Villegas, Eladio Gutierrez, and Oscar Plata. Accelerating time series motif discovery in the Intel Xeon Phi KNL processor. *The Journal of Supercomputing*, Jun 2019. URL: https://doi.org/10.1007/s11227-019-02923-5, https://doi.org/10.1007/s11227-019-02923-5. (Cited on pages 30, 39, 66 and 159)

[72] Mikhail Zymbler, Andrey Polyakov, and Mikhail Kipnis. Time series discord discovery on intel many-core systems. In Leonid Sokolinsky and Mikhail Zymbler, editors, *Parallel Computational Technologies*, pages 168–182, Cham, 2019. Springer International Publishing. (Cited on pages 30 and 66)

[73] Dieter De Paepe, Sander Vanden Hautte, Bram Steenwinckel, Filip De Turck, Femke Ongenae, Olivier Janssens, and Sofie Van Hoecke. A generalized matrix profile framework with support for contextual series analysis. *Engineering Applications of Artificial Intelligence*, 90:103487, 2020. (Cited on page 30)

[74] Yan Zhu, Shaghayegh Gharghabi, Diego Furtado Silva, Hoang Anh Dau, Chin-Chia Michael Yeh, Nader Shakibay Senobari, Abdulaziz Almaslukh, Kaveh Kamgar, Zachary Zimmerman, Gareth Funning, Abdullah Mueen, and Eamonn Keogh. The swiss army knife of time series data mining: ten useful things you can do with the matrix profile and ten lines of code. *Data Mining and Knowledge Discovery*, 34(4):949–979,

2020. URL: https://doi.org/10.1007/s10618-019-00668-6, https://doi.org/10.1007/s10618-019-00668-6. (Cited on page 30)

[75] Yan Zhu, Chin-Chia Michael Yeh, Zachary Zimmerman, and Eamonn Keogh. Matrix profile xvii: Indexing the matrix profile to allow arbitrary range queries. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1846–1849. IEEE, 2020. (Cited on page 30)

[76] Reza Akbarinia and Bertrand Cloez. Efficient matrix profile computation using different distance functions. *arXiv preprint arXiv:1901.05708*, 2019. (Cited on page 31)

[77] Michele Linardi, Yan Zhu, Themis Palpanas, and Eamonn Keogh. Matrix profile goes mad: variable-length motif and discord discovery in data series. *DATA MINING AND KNOWLEDGE DISCOVERY*, 2020. (Cited on page 31)

[78] Frank Madrid, Shima Imani, Ryan Mercer, Zachary Zimmerman, Nader Shakibay, and Eamonn Keogh. Matrix profile xx: Finding and visualizing time series motifs of all lengths using the matrix profile. In *2019 IEEE International Conference on Big Knowledge (ICBK)*, pages 175–182. IEEE, 2019. (Cited on page 31)

[79] Piotr Przymus and Krzysztof Kaczmarski. Time series queries processing with GPU support. In *New Trends in Databases and Information Systems*, pages 53–60, Cham, 2014. Springer International Publishing. (Cited on page 31)

[80] Piotr Przymus and Krzysztof Kaczmarski. Time series queries processing with gpu support. In *New trends in databases and information systems*, pages 53–60. Springer, 2014. (Cited on page 31)

[81] Miquel L Alomar, Vincent Canals, Nicolas Perez-Mora, Víctor Martínez-Moll, and Josep L Rosselló. Fpga-based stochastic echo state networks for time-series forecasting. *Computational intelligence and neuroscience*, 2016, 2016. (Cited on page 31)

[82] Ana Dalia Pano-Azucena, Esteban Tlelo-Cuautle, Luis Gerardo de la Fraga, Carlos Sanchez-Lopez, JJ Rangel-Magdaleno, and Sheldon X-D Tan. Prediction of chaotic time-series with different mle values using fpga-based anns. In *2017 14th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, pages 1–4. IEEE, 2017. (Cited on page 31)

[83] Ana Dalia Pano-Azucena, Esteban Tlelo-Cuautle, Sheldon X-D Tan, and Luis Gerardo De la Fraga. Fpga-based implementation of a multilayer perceptron suitable for chaotic time series prediction. *Technologies*, 6(4):90, 2018. (Cited on page 31)

[84] Xinzhe Zang, Zhenbin Gao, Mengyuan Li, and Xia Wang. Fpga implementation of pulse coupled neural network on for time series of an image. In *Proceedings of the 2018 International Conference on Electronics and Electrical Engineering Technology*, pages 212–216, 2018. (Cited on page 31)

[85] Jun Liu, Jiasheng Wang, Yu Zhou, and Fang Liu. A cloud server oriented fpga accelerator for lstm recurrent neural network. *IEEE Access*, 7:122408–122418, 2019. (Cited on page 31)

[86] S. Borzsony, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings 17th International Conference on Data Engineering*, pages 421–430, 2001. https://doi.org/10.1109/ICDE.2001.914855. (Cited on pages 31, 32, 109 and 162)

[87] Ximeng Liu, Rongxing Lu, Jianfeng Ma, Le Chen, and Haiyong Bao. Efficient and privacy-preserving skyline computation framework across domains. *Future Generation Computer Systems*, 62:161–174, 2016. (Cited on pages 32 and 162)

[88] Alfredo Cuzzocrea, Panagiotis Karras, and Akrivi Vlachou. Effective and efficient skyline query processing over attribute-order-preserving-free encrypted data in cloud-enabled databases. *Future Generation Computer Systems*, 126:237–251, 2022. (Cited on pages 32 and 163)

[89] Helan Liang, Bincheng Ding, Yanhua Du, and Fanzhang Li. Parallel optimization of qos-aware big service processes with discovery of skyline services. *Future Generation Computer Systems*, 125:496–514, 2021. (Cited on pages 32 and 163)

[90] Shangguang Wang, Lin Huang, Lei Sun, Ching-Hsien Hsu, and Fangchun Yang. Efficient and reliable service selection for heterogeneous distributed software systems. *Future Generation Computer Systems*, 74:158–167, 2017. (Cited on pages 32 and 163)

[91] Babar Shahzaad, Athman Bouguettaya, Sajib Mistry, and Azadeh Ghari Neiat. Resilient composition of drone services for delivery. *Future Generation Computer Systems*, 115:335–350, 2021. (Cited on pages 32 and 163)

[92] Hongbing Wang, Xingguo Hu, Qi Yu, Mingzhu Gu, Wei Zhao, Jia Yan, and Tianjing Hong. Integrating reinforcement learning and skyline computing for adaptive service composition. *Information Sciences*, 519:141–160, 2020. URL: https://www.sciencedirect.com/science/article/pii/S0020025520300414, https://doi.org/https://doi.org/10.1016/j.ins.2020.01.039. (Cited on pages 32 and 163)

[93] Yandong Zheng, Rongxing Lu, Beibei Li, Jun Shao, Haomiao Yang, and Kim-Kwang Raymond Choo. Efficient privacy-preserving data merging and skyline computation over multi-source encrypted data. *Information Sciences*, 498:91–105, 2019. URL: https://www.sciencedirect.com/science/article/pii/S0020025519304591, https://doi.org/https://doi.org/10.1016/j.ins.2019.05.055. (Cited on pages 32 and 163)

[94] Wajdi Dhifli, Nour El Islem Karabadji, and Mohamed Elati. Evolutionary mining of skyline clusters of attributed graph data. *Information Sciences*, 509:501–514, 2020. URL: https://www.sciencedirect.com/science/article/pii/S0020025518307655, https://doi.org/https://doi.org/10.1016/j.ins.2018.09.053. (Cited on pages 32 and 163)

[95] Kenneth S Bøgh, Sean Chester, Darius Šidlauskas, and Ira Assent. Template skycube algorithms for heterogeneous parallelism on multicore and gpu architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 447–462, 2017. (Cited on pages 32 and 110)

[96] Kian-Lee Tan, Pin-Kwang Eng, Beng Chin Ooi, et al. Efficient progressive skyline computation. In *VLDB*, volume 1, pages 301–310, 2001. (Cited on page 32)

[97] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive skyline computation in database systems. *ACM Transactions on Database Systems (TODS)*, 30(1):41–82, 2005. (Cited on page 32)

[98] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. Skyline with presorting. In *ICDE*, volume 3, pages 717–719, 2003. (Cited on page 33)

[99] Ken CK Lee, Baihua Zheng, Huajing Li, and Wang-Chien Lee. Approaching the skyline in z order. In *VLDB*, volume 7, pages 279–290, 2007. (Cited on page 33)

[100] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. Efficient sort-based skyline evaluation. *ACM Transactions on Database Systems (TODS)*, 33(4):1–49, 2008. (Cited on page 33)

[101] Shiming Zhang, Nikos Mamoulis, and David W Cheung. Scalable skyline computation using object-based space partitioning. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 483–494, 2009. (Cited on page 33)

[102] Jongwuk Lee and Seung-won Hwang. Scalable skyline computation using a balanced pivot selection technique. *Information Systems*, 39:1–21, 2014. (Cited on page 33)

[103] Sungwoo Park, Taekyung Kim, Jonghyun Park, Jinha Kim, and Hyeonseung Im. Parallel skyline computation on multicore architectures. In *2009 IEEE 25th International Conference on Data Engineering*, pages 760–771. IEEE, 2009. (Cited on page 33)

[104] Wonik Choi, Ling Liu, and Boseon Yu. Multi-criteria decision making with skyline computation. In *2012 IEEE 13th International Conference on Information Reuse & Integration (IRI)*, pages 316–323. IEEE, 2012. (Cited on page 33)

[105] Kenneth S Bøgh, Ira Assent, and Matteo Magnani. Efficient gpu-based skyline computation. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, pages 1–6, 2013. (Cited on page 33)

[106] Yi-Wen Peng and Wei-Mei Chen. Parallel k-dominant skyline queries in high-dimensional datasets. *Information Sciences*, 496:538–552, 2019. URL: https://www.sciencedirect.com/science/article/pii/S0020025519300490, https://doi.org/https://doi.org/10.1016/j.ins.2019.01.039. (Cited on page 33)

[107] Xuemin Lin, Yidong Yuan, Wei Wang, and Hongjun Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *21st International Conference on Data Engineering (ICDE'05)*, pages 502–513. IEEE, 2005. (Cited on page 34)

[108] Yufei Tao and Dimitris Papadias. Maintaining sliding window skylines on data streams. *IEEE Transactions on Knowledge and Data Engineering*, 18(3):377–391, 2006. (Cited on pages 34, 110 and 164)

[109] Michael Morse, Jignesh M Patel, and William I Grosky. Efficient continuous skyline computation. *Information Sciences*, 177(17):3411–3437, 2007. (Cited on pages 34, 110 and 164)

[110] Tiziano De Matteis, Salvatore Di Girolamo, and Gabriele Mencagli. Continuous skyline queries on multicore architectures. *Concurrency and Computation: Practice and Experience*, 28(12):3503–3522, 2016. (Cited on pages 34, 110 and 164)

[111] Hua Lu, Yongluan Zhou, and Jonas Haustad. Efficient and scalable continuous skyline monitoring in two-tier streaming settings. *Information Systems*, 38(1):68–81, 2013. (Cited on page 34)

[112] Shengli Sun, Zhenghua Huang, Hao Zhong, Dongbo Dai, Hongbin Liu, and Jinjiu Li. Efficient monitoring of skyline queries over distributed data streams. *Knowledge and information systems*, 25(3):575–606, 2010. (Cited on page 34)

[113] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M Badia, Xavier Martorell, Eduard Ayguade, and Jesus Labarta. Productive programming of gpu clusters with ompss. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 557–568. IEEE, 2012. (Cited on page 35)

[114] Mehmet E Belviranli, Laxmi N Bhuyan, and Rajiv Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–20, 2013. (Cited on page 35)

[115] Prasanna Vasant Pandit. *Cooperative Execution of Opencl Programs on Multiple Heterogeneous Devices*. PhD thesis, Indian Institute of Science, 2018. (Cited on page 35)

[116] Intel Corp. Intel® oneapi programming guide. https://www.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top.html, 2021. (Cited on page 35)

[117] Angeles Navarro, Francisco Corbera, Andres Rodriguez, Antonio Vilches, and Rafael Asenjo. Heterogeneous parallel_for template for CPU–GPU chips. *International Journal of Parallel Programming*, 47(2):213–233, Apr 2019. URL: https://doi.org/10.1007/s10766-018-0555-0, https://doi.org/10.1007/s10766-018-0555-0. (Cited on pages 35, 40, 46 and 159)

[118] R. Kaleem, R. Barik, T. Shpeisman, C. Hu, B. T. Lewis, and K. Pingali. Adaptive heterogeneous scheduling for integrated GPUs. In *Intl Conf on Parallel Architecture and Compilation Techniques (PACT)*, pages 151–162, Aug 2014. https://doi.org/10.1145/2628071.2628088. (Cited on page 35)

[119] Borja Pérez, José Luis Bosque, and Ramón Beivide. Simplifying programming and load balancing of data parallel applications on Heterogeneous Systems. In *W. on General Purpose Processing Using GPU*, GPGPU '16, pages 42–51, 2016. URL: http://doi.acm.org/10.1145/2884045.2884051, https://doi.org/10.1145/2884045.2884051. (Cited on page 35)

[120] Mehmet Belviranli, Laxmi Bhuyan, and Rajiv Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4), 01 2013. https://doi.org/10.1145/2400682.2400716. (Cited on page 36)

[121] Zoran Jakšić, Nicola Cadenelli, David Buchaca Prats, Jordà Polo, Josep Lluís [Berral Garcia], and David Carrera Perez. A highly parameterizable framework for conditional restricted boltzmann machine based workloads accelerated with fpgas and opencl. *Future Generation Computer Systems*, 104:201 – 211, 2020. URL: http://www.sciencedirect.com/science/article/pii/S0167739X19313676, https://doi.org/https://doi.org/10.1016/j.future.2019.10.025. (Cited on page 36)

[122] C. Christ, B. Nord, K. Gozman, and K. Ottenbreit. Fast identification of strong gravitational lenses with deep learning on FPGAs and other heterogeneous computing devices. In *American Astronomical Society Meeting Abstracts*, American Astronomical Society Meeting Abstracts, page 303.03, January 2020. (Cited on page 36)

[123] B. Ringlein, F. Abel, A. Ditter, B. Weiss, C. Hagleitner, and D. Fey. Zrlmpi: A unified programming model for reconfigurable heterogeneous computing clusters. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 220–220, 2020. (Cited on page 36)

[124] M. Bacis, R. Brondolin, and M. D. Santambrogio. Blastfunction: an fpga-as-a-service system for accelerated serverless computing. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 852–857, 2020. (Cited on page 36)

[125] Yao Chen, Xin Long, Jiong He, Yuhang Chen, Hongshi Tan, Zhenxiang Zhang, Marianne Winslett, and Deming Chen. Haocl: Harnessing large-scale heterogeneous processors made easy, 2020. https://arxiv.org/abs/2005.08466. (Cited on page 36)

[126] Xinyu Chen, Yao Chen, Ronak Bajaj, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. Is fpga useful for hash joins? In *CIDR*, page n/a, 2020. (Cited on page 36)

[127] K. Hasegawa, R. Ishikawa, M. Nishizawa, K. Kawamura, M. Tawada, and N. Togawa. Fpga-based heterogeneous solver for three-dimensional routing. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 11–12, 2020. (Cited on page 36)

[128] Xiaoyu Hou, Daying Quan, Wei Fan, Xiaoping Jin, and Hengliang Liu. Implementation of SAR echo simulation algorithm on heterogeneous embedded computing platform. *IOP Conference Series: Materials Science and Engineering*, 719:012031, jan 2020. https://doi.org/10.1088/1757-899x/719/1/012031. (Cited on page 36)

[129] Jiajian Xiao, Philipp Andelfinger, Wentong Cai, Paul Richmond, Alois Knoll, and David Eckhoff. Openablext: An automatic code generation framework for agent-based simulations on cpu-gpu-fpga heterogeneous platforms. *Concurrency and Computation: Practice and Experience*, n/a(n/a):e5807, 2020. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5807, https://arxiv.org/abs/https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5807, https://doi.org/10.1002/cpe.5807. (Cited on page 36)

[130] Mohammad Hosseinabady and Jose Nunez-Yanez. Sparse matrix-dense matrix multiplication on heterogeneous cpu+fpga embedded system. In *Proceedings of the 11th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures / 9th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM'2020, page n/a, New York, NY, USA, 2020. Association for Computing Machinery. URL: https://doi.org/10.1145/3381427.3381428, https://doi.org/10.1145/3381427.3381428. (Cited on page 36)

[131] Mohammadreza Soltaniyeh, Richard P Martin, and Santosh Nagarakatte. Synergistic cpu-fpga acceleration of sparse linear algebra. *arXiv preprint arXiv:2004.13907*, 2020. (Cited on page 37)

[132] Marco Carreras, Gianfranco Deriu, Luigi Raffo, Luca Benini, and Paolo Meloni. Optimizing temporal convolutional network inference on fpga-based accelerators, 2020. https://arxiv.org/abs/2005.03775. (Cited on page 37)

[133] Nicola Cadenelli, Zoran Jaksic, Jorda Polo, and David Carrera. Considerations in using opencl on gpus and fpgas for throughput-oriented genomics workloads. *Future Generation Computer Systems*, 94:148 – 159, 2019. URL: http://www.sciencedirect.com/science/article/pii/S0167739X18314183, https://doi.org/https://doi.org/10.1016/j.future.2018.11.028. (Cited on page 37)

[134] Ryohei Kobayashi, Norihisa Fujita, Yoshiki Yamaguchi, Ayumi Nakamichi, and Taisuke Boku. Opencl-enabled gpu-fpga accelerated computing with inter-fpga communication. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops*, HPCAsia2020, page 17–20, New York, NY, USA, 2020. Association for Computing Machinery. URL: https://doi.org/10.1145/3373271.3373275, https://doi.org/10.1145/3373271.3373275. (Cited on page 37)

[135] Reza Ramezani. Dynamic scheduling of task graphs in multi-fpga systems using critical path. *The Journal of Supercomputing*, 2020. URL: https://doi.org/10.1007/s11227-020-03281-3, https://doi.org/10.1007/s11227-020-03281-3. (Cited on page 37)

[136] Maria Angelica Davila Guzman, Raúl Nozal, Rubén Gran Tejero, Maria Villarroya-Gaudo, Darío Suárez Gracia, and Jose Luis Bosque. Cooperative cpu, gpu, and fpga heterogeneous execution with enginecl. *The Journal of Supercomputing*, 75(3):1732–1746, 2019. (Cited on page 37)

[137] Y. Meng, S. Kuppannagari, and V. Prasanna. Accelerating proximal policy optimization on cpu-fpga heterogeneous platforms. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 19–27, 2020. (Cited on page 37)

[138] Peilun Du, Zichang Sun, Haitao Zhang, and Huadong Ma. Feature-aware task scheduling on cpu-fpga heterogeneous platforms. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 534–541. IEEE, 2019. (Cited on page 37)

[139] Hanqing Zeng and Viktor Prasanna. Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 255–265, 2020. (Cited on page 37)

[140] Jason Cong, Zhenman Fang, Yao Hu, and Di Wu. K-flow: A programming and scheduling framework to optimize dataflow execution on cpu-fpga platforms. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 287–287, 2018. (Cited on page 37)

[141] W. Zhang, X. Liao, and H. Jin. Fine-grained scheduling in fpga-based convolutional neural networks. In *2020 IEEE 5th International Conference*

on Cloud Computing and Big Data Analytics (ICCCBDA), pages 120–128, 2020. (Cited on page 37)

[142] A. Al-Zoubi and K. Tatas. Rapid high-level fpga resource estimation for a novel heterogeneous platform scheduling scheme. In *2020 11th International Conference on Information and Communication Systems (ICICS)*, pages 378–381, 2020. (Cited on page 37)

[143] Abdessamad Ait El Cadi, Omar Souissi, Rabie Ben Atitallah, Nicolas Belanger, and Abdelhakim Artiba. Mathematical programming models for scheduling in a cpu/fpga architecture with heterogeneous communication delays. *Journal of Intelligent Manufacturing*, 29(3):629–640, 2018. (Cited on page 37)

[144] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. Heterogeneous resource-elastic scheduling for cpu+ fpga architectures. In *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, pages 1–6, 2019. (Cited on page 37)

[145] Jose Nunez-Yanez, Sam Amiri, Mohammad Hosseinabady, Andrés Rodríguez, Rafael Asenjo, Angeles Navarro, Dario Suarez, and Ruben Gran. Simultaneous multiprocessing in a software-defined heterogeneous fpga. *The Journal of Supercomputing*, Apr 2018. URL: https://doi.org/10.1007/s11227-018-2367-9, https://doi.org/10.1007/s11227-018-2367-9. (Cited on page 37)

[146] Angeles Navarro, Francisco Corbera, Andres Rodriguez, Antonio Vilches, and Rafael Asenjo. Heterogeneous parallel_for template for CPU–GPU chips. *International Journal of Parallel Programming*, Jan 2018. (Cited on pages 37, 73, 77, 83, 91, 92 and 97)

[147] Andrés Rodríguez, Angeles Navarro, Rafael Asenjo, Francisco Corbera, Rubén Gran, Darío Suárez, and Jose Nunez-Yanez. Exploring heterogeneous scheduling for edge computing with cpu and fpga mpsocs. *Journal of Systems Architecture*, 98:27 – 40, 2019. URL: http://www.sciencedirect.com/science/article/pii/S1383762119300918, https://doi.org/https://doi.org/10.1016/j.sysarc.2019.06.006. (Cited on page 37)

[148] Clara E Yoon, Ossian O'Reilly, Karianne J Bergen, and Gregory C Beroza. Earthquake detection through computationally efficient similarity search. *Science advances*, 1(11), 12 2015. URL: https://www.ncbi.nlm.nih.gov/pubmed/26665176, https://doi.org/10.1126/sciadv.1501057. (Cited on page 39)

[149] Joshua D. Rhodes, Wesley J. Cole, Charles R. Upshaw, Thomas F Edgar, and Michael Webber. Clustering analysis of residential electricity demand profiles. *Applied Energy*, 135:461–471, 12 2014. https://doi.org/10.1016/j.apenergy.2014.08.111. (Cited on page 39)

[150] Ilya Kolb, Giovanni Talei Franzesi, Michael Wang, Suhasa B. Kodandaramaiah, Craig R. Forest, Edward S. Boyden, and Annabelle C. Singer. Evidence for long-timescale patterns of synaptic inputs in ca1 of awake behaving mice. *J. Neuroscience*, 38(7):1821–1834, 2018. URL: http://www.jneurosci.org/content/38/7/1821, https://arxiv.org/abs/http://www.jneurosci.org/content/38/7/1821.full.pdf, https://doi.org/10.1523/JNEUROSCI.1519-17.2017. (Cited on page 39)

[151] Szigeti Balázs, Deogade Ajinkya, and Webb Barbara. Searching for motifs in the behaviour of larval drosophila melanogaster and caenorhabditis elegans reveals continuity between behavioural states. *J. The Royal Society Interface*, 12(113), 2015. URL: https://doi.org/10.1098/rsif.2015.0899, https://doi.org/10.1098/rsif.2015.0899. (Cited on page 39)

[152] Michael Voss, Rafael Asenjo, and James Reinders. *Pro TBB: C++ Parallel Programming with Threading Building Blocks*. Apress, 2019. (Cited on pages 39, 45 and 159)

[153] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5), 1999. (Cited on page 45)

[154] Stephen Junkins. The compute architecture of Intel Processor Graphics Gen9. https://software.intel.com/en-us/file/the-compute-architecture-of-intel-processor-graphics-gen9-v1d0pdf, 2015. Accessed: 2019-09. (Cited on page 49)

[155] Khronos. OpenCL Atomic functions. https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/atom_cmpxchg.html. Accessed: 2019-05-01. (Cited on page 51)

[156] Charles R. Nelson and Charles R. Plosser. Trends and random walks in macroeconomic time series: Some evidence and implications. *Journal of Monetary Economics*, 10(2):139 – 162, 1982. URL: http://www.sciencedirect.com/science/article/pii/0304393282900125, https://doi.org/https://doi.org/10.1016/0304-3932(82)90012-5. (Cited on page 51)

[157] Eamonn Keogh. The UCR Matrix Profile Page. https://www.cs.ucr.edu/~eamonn/MatrixProfile.html, 2016. Accessed: 2019-09-13. (Cited on page 56)

[158] Gabriel Pfeilschifter. Time series analysis with matrix profile on hpc systems. Masterarbeit, Technische Universität München, 2019. (Cited on page 69)

[159] Mikhail Zymbler, Andrey Polyakov, and Mikhail Kipnis. Time series discord discovery on intel many-core systems. In Leonid Sokolinsky and Mikhail Zymbler, editors, *Parallel Computational Technologies*, pages 168–182, Cham, 2019. Springer International Publishing. (Cited on page 69)

[160] Ivan Fernandez, Alejandro Villegas, Eladio Gutierrez, and Oscar Plata. Accelerating time series motif discovery in the intel xeon phi knl processor. *The Journal of Supercomputing*, 75(11):7053–7075, 2019. (Cited on page 69)

[161] Intel. Intel FPGA Best Practices Guide. https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807516407.html#mwh1391807501939, 2020. Accessed: 2020-07-03. (Cited on pages 74, 88 and 92)

[162] Intel. Intel FPGA Programming Guide. https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html, 2020. Accessed: 2020-07-03. (Cited on page 76)

[163] A. Navarro, A. Vilches, F. Corbera, and R. Asenjo. Strategies for maximizing utilization on multi-CPU and multi-GPU heterogeneous architectures. *The Journal of Supercomputing*, 70(2):756–771, 2014. (Cited on page 79)

[164] Zhenning Wang, Long Zheng, Quan Chen, and Minyi Guo. CPU+GPU scheduling with asymptotic profiling. *Parallel Computing*, 2:107–115, feb 2014. (Cited on page 79)

[165] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO 42*, pages 45–55. ACM, 2009. https://doi.org/10.1145/1669112.1669121. (Cited on page 79)

[166] Antonio Vilches. Stratixmonitorlib, July 2020. URL: https://doi.org/10.5281/zenodo.3948283, https://doi.org/10.5281/zenodo.3948283. (Cited on pages 91 and 102)

[167] Charles R Nelson and Charles R Plosser. Trends and random walks in macroeconmic time series: some evidence and implications. *Journal of monetary economics*, 10(2):139–162, 1982. (Cited on page 91)

[168] D.C. Rudolph and C.D. Polychronopoulos. An efficient message-passing scheduler based on guided self scheduling. In *3rd Intl. Conf. on Supercomputing*, ICS'89, 1989. (Cited on page 91)

[169] Intel. oneapi webpage. https://www.oneapi.com, July 2020. Accessed: 2020-07-15. (Cited on page 104)

[170] Denisa-Andreea Constantinescu, Angeles Navarro, Francisco Corbera, Juan-Antonio Fernández-Madrigal, and Rafael Asenjo. Efficiency and productivity for decision making on low-power heterogeneous cpu+ gpu socs. *The Journal of Supercomputing*, pages 1–22, 2020. (Cited on page 105)

[171] Xiaoyong Li, Yijie Wang, Xiaoling Li, and Yuan Wang. Parallelizing skyline queries over uncertain data streams with sliding window partitioning and grid index. *Knowledge and Information Systems*, 41(2):277–309, 2014. (Cited on pages 110 and 164)

[172] Karim Alami, Nicolas Hanusse, Patrick Kamnang-Wanko, and Sofian Maabout. The negative skycube. *Information Systems*, 88:101443, 2020. (Cited on page 110)

[173] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001. (Cited on page 113)

[174] Rob Farber. *CUDA application design and development*. Elsevier, 2011. (Cited on page 113)

[175] Data parallel compatibility tool. https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compatibility-tool.html, 2022. Accessed: 2022-01-16. (Cited on page 120)

[176] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002. https://doi.org/10.1109/71.993206. (Cited on page 126)

[177] Chrono library. https://en.cppreference.com/w/cpp/chrono, 2022. Accessed: 2022-04-05. (Cited on page 134)

[178] Dataset house. http://usa.ipums.org/usa/, 2021. Accessed: 2021-09-20. (Cited on page 135)

[179] Dataset nba. http://databasebasketball.com, 2021. Accessed: 2021-09-20. (Cited on page 135)

[180] Dataset covertype. http://archive.ics.uci.edu/ml/datasets/Covertype, 2021. Accessed: 2021-09-20. (Cited on page 135)

[181] Dataset weather. http://cru.uea.ac.uk/cru/data/hrg/tmc/, 2021. Accessed: 2021-09-20. (Cited on page 135)