UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA

TIAGO KNORST

Collaborative-Aware CPU *Thread Throttling* and FPGA *HLS-Versioning*

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Microeletronics

Advisor: Prof. Dr. Antonio Carlos Schneider
Beck Filho
Coadvisor: Prof. Dr. Mateus Beck Rutzig

Porto Alegre
September 2022

CIP — CATALOGING-IN-PUBLICATION

Knorst, Tiago

Collaborative-Aware CPU *Thread Throttling* and FPGA *HLS-Versioning* / Tiago Knorst. – Porto Alegre: PGMICRO da UFRGS, 2022.

98 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2022. Advisor: Antonio Carlos Schneider Beck Filho; Coadvisor: Mateus Beck Rutzig.

1. CPU-GPU. 2. CPU-FPGA. 3. Throttling. 4. Collaborative. 5. TLP. 6. HLS. I. Schneider Beck Filho, Antonio Carlos. II. Beck Rutzig, Mateus. III. Título.

“Doubt is not a pleasant condition,
but certainty is absurd.”
— VOLTAIRE

AGRADECIMENTOS

A realização deste trabalho teve a colaboração de diversas pessoas, às quais merecem os devidos agradecimentos.

Agradeço, primeiramente, aos meus pais Flávio e Inês e à minha irmã Gabriela pelo amor, carinho e apoio incondicional, especialmente durante estes dois anos de ensino remoto.

Agradeço à minha namorada Amanda Mohr, que tenho a sorte de estar ao meu lado me encorajando na busca pelos meus objetivos.

Agradeço ao meu orientador deste trabalho, Professor Antonio Carlos Schneider Beck Filho, pela paciência, conselhos e ensinamentos passados. Devo agradecer, também, ao meu co-orientador, Professor Mateus Beck Rutzig, pela enorme contribuição na realização deste trabalho e em meu ingresso no Mestrado.

Agradeço a todos os meus colegas de laboratório, especialmente ao Michael Guilherme Jordan, que teve grande contribuição, tanto técnica quanto científica, para a construção deste e outros trabalhos.

Agradeço à CAPES, pelo auxílio financeiro. Também, aos professores e funcionários do Programa de Pós-Graduação em Microeletrônica e do Instituto de Informática, que oferecem às condições para o ensino de excelência provido na Universidade Federal do Rio Grande do Sul.

ABSTRACT

Warehouses and Cloud Servers have been adopting collaborative CPU-GPU and CPU-FPGA architectures as alternatives to enable extra acceleration for applications by partitioning threads/kernels execution across both devices. However, exploiting the benefits of these environments is challenging, since there are many factors that may influence performance and energy consumption, such as the number of CPU threads, the workload balance, and optimization techniques such as FPGA *HLS (High-Level Synthesis)-versioning*. This work shows that maximizing resource utilization by triggering the highest number of CPU threads does not always result in the best efficiency for both CPU-GPU and CPU-FPGA architectures. Moreover, our experiments show that the amount of data distributed to each device (workload balance) affects the needed CPU processing power and, therefore, the number of active CPU threads for the application. To address these problems, we first propose ETCG – Energy-aware CPU *Thread Throttling* for CPU-GPU collaborative environments. ETCG transparently selects a near-optimal number of CPU threads to minimize the energy-delay product (EDP) of CPU-GPU applications. In the second study, we propose ETCF – Energy-Aware CPU *Thread Throttling* and Workload Balancing Framework for CPU-FPGA collaborative environments. ETCF automatically provides efficient CPU-FPGA execution by selecting the right workload balance and the number of CPU threads for a given collaborative application. ETCF framework offers different optimization goals: performance, energy, or EDP. Compared to the baseline (an equally balanced workload executing with the maximum number of CPU threads), ETCG and ETCF provide, on average, 73% and 93% of EDP reduction, respectively. We also show that both ETCG and ETCF achieve near-optimal solutions by comparing it to an exhaustive search, but just taking up to 5% of its searching time. Finally, in the third study, we considered a task-collaborative CPU-FPGA environment. We investigate the impact of collaboratively applying *Thread Throttling* on the CPU side and *HLS-versioning* on the FPGA side. We use a multi-tenant Cloud service as our object of study, where sequence of application requests with different priorities result in DAGs of application kernels that must be executed over the heterogeneous architecture. We show that by synergistically applying *Thread Throttling* and *HLS-Versioning* to the incoming kernels may improve the EDP in up to 41x over the non-optimized execution.

Keywords: CPU-GPU. CPU-FPGA. throttling. collaborative. TLP. HLS.

CPU *Thread Throttling* e FPGA HLS-*versioning* aplicados colaborativamente

RESUMO

Servidores em Nuvem vem adotando arquiteturas colaborativas de CPU-GPU e CPU-FPGA como alternativas para permitir aceleração extra às aplicações através do particionando da execução de *threads/kernels* em ambos os dispositivos. No entanto, explorar os benefícios destes ambientes é desafiador, pois existem muitos fatores que podem influenciar o desempenho e o consumo de energia, como o número de *threads* da CPU, o balanceamento da carga de trabalho (*Worload Balance*) e técnicas de otimização como FPGA HLS (*High-Level Synthesis*)-*versioning*. Este trabalho mostra que maximizar a utilização de recursos acionando o maior número de threads de CPU nem sempre resulta na melhor eficiência tanto em arquiteturas CPU-GPU quanto CPU-FPGA. Além disso, os experimentos mostram que a quantidade de dados distribuídos para cada dispositivo (*Workload Balance*) afeta o poder de processamento necessário da CPU e, portanto, o número ótimo de *threads* de CPU para a execução da aplicação. Com o intuito de otimizar a execução de aplicações colaborativas CPU-GPU, inicialmente propomos o ETCG – *Energy-aware CPU Thread Throttling for CPU-GPU collaborative environments*. O ETCG é capaz de selecionar de forma transparente um número quase ótimo de *threads* na CPU visando a minimizar o produto entre atraso e energia consumida (EDP) de aplicações CPU-GPU. No segundo estudo, propomos ETCF – *Energy-Aware CPU Thread Throttling and Workload Balancing Framework* para ambientes colaborativos CPU-FPGA. O ETCF fornece uma execução CPU-FPGA eficiente ao selecionar apropriadamente o Workload Balance e o número de *threads* na CPU para uma aplicação colaborativa. Além disso, são oferecidos diferentes objetivos de otimização: desempenho, energia ou EDP. Em comparação à linha de base (considerando uma carga de trabalho igualmente equilibrada entre os dispositivos e usando o número máximo de *threads* na CPU), nossos experimentos mostram que ETCG e ETCF fornecem, em média, 73% e 93% de redução de EDP, respectivamente. Também mostramos que tanto o ETCG quanto o ETCF alcançam soluções quase ótimas comparando-os a uma busca exaustiva, mas levando apenas 5% de seu tempo de busca no pior cenário. Finalmente, no terceiro estudo, consideramos um ambiente de tarefas colaborativas CPU-FPGA. Investigamos o impacto de aplicar sinergicamente *Thread Throttling* no lado da CPU e do *HLS-Versioning* no lado do FPGA. Utilizamos como objeto de estudo um serviço de nuvem multi-inquilino, onde a sequência de requisições de aplicações com diferentes prioridades resulta em DAGs de *kernels* de aplicações que devem ser executados sobre a arquitetura heterogênea. Nossos experimentos mostram que, ao aplicar sinergicamente *Thread Throttling* e *HLS-Versioning* aos *kernels* recebidos, pode-se melhorar o EDP em até 41x em relação à execução não otimizada.

Palavras-chave: CPU-GPU. CPU-FPGA. *throttling*. colaborativo. TLP. HLS.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

CPU	Central Processing Unit
CMP	Chip Multiprocessor
DAG	Direct Acyclic Graph
DSE	Design Space Exploration
EDP	Energy-Delay Product
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
HLS	High-Level Systensis
OS	Operating System
SMT	Simultaneous Multithreading
TLP	Thread-level Parallelism
TT	Thread Throttling
WB	Workload Balance

CONTENTS

1 INTRODUCTION

Computation is increasingly moving away from traditional data servers to the cloud, leveraging cost savings, reliability, and scalability. Cloud Service Providers such as Amazon, Google, and Microsoft provide on-demand access to hardware resources and services to their costumers (WU et al., 2010). To deliver that, vendors host and manage Cloud Data Warehouses, which maintain data and offered applications. Current Cloud services usually comprise the use of applications such as machine learning, big data, video, and audio processing, which impose high performance requirements. However, to reach these requirements while meeting power and energy constraints, inherent in such systems, clever strategies are required from hardware designers.

Hardware designing have taken advantage of the tremendous advances in microelectronics since Complementary Metal–Oxide Semiconductor technology became widespread, which enabled an increase in computers' efficiency, following Moore's and Dennard's scaling laws (BECK; LISBÔA; CARRO, 2012). With transistors enhancements reaching physical limits, though, both laws came to an end, imposing the challenge of achieving the performance requirements while meeting energy constraints. One of the main strategies to reach it is by exploiting parallelism from control and data-oriented applications, and properly utilizing the hardware resources to benefit from such exploitation.

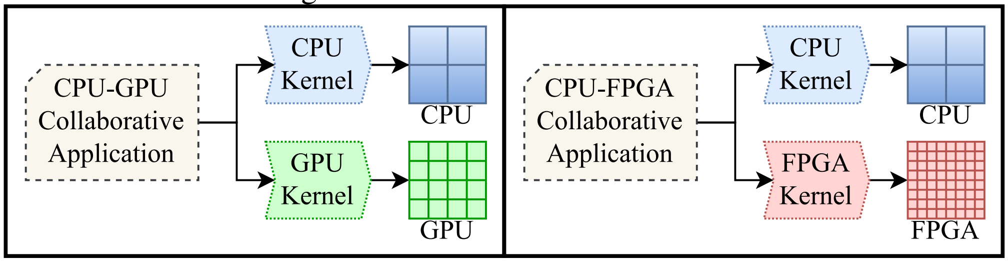
Thread-level Parallelism (TLP) exploitation is commonly employed in Chip Multiprocessors (CMPs), which traditionally equip Cloud Data Warehouses. TLP exploitation takes advantage of the multiple cores inside a single chip offered by CMPs to execute portions of the program (threads) in different cores simultaneously, increasing concurrency and improving performance in parallel applications. To further optimize the efficiency in Cloud Data Warehouses, vendors moved their systems from traditional homogeneous CPU-only systems to heterogeneous CPU-GPU and CPU-FPGA systems - by coupling a Graphics Processing Units (GPU) and a Field-Programmable Gate Array (FPGA), respectively -, which enable extra acceleration when executing the applications on the most suitable device (KACHRIS; SOUDRIS, 2016).

In the last two decades, GPUs gained attractive because of their capability to accelerate graphical workloads compared to traditional CPUs. Despite being developed originally to deal with graphic applications, the creation and disposal of CUDA and OpenCL made it possible to use GPUs for general-purpose parallel computing, thereby allowing the acceleration of massive data-parallel applications (DU et al., 2012).

Similarly, the use of FPGAs became a complementary alternative due to the implementation flexibility offered by High-Level Synthesis, which allows programmers to express their designs with high-level programming languages (STONE; GOHARA; SHI, 2010). This way, the use of FPGA provides opportunities for further performance and energy improvements, once it can cover other software behaviors not covered by CPUs and GPUs, such as bit-level parallelism and low-precision arithmetic, still keeping lower energy consumption levels (CRAVEN; ATHANAS, 2007).

To exploit both devices, conventional heterogeneous applications offload all their compute-intensive tasks to the GPU or the FPGA, once these devices are more efficient than CPUs in data-parallel workloads, leading the host CPU to remain idle while these portions of the application are executed. To fully utilize the hardware resources, Collaborative Computing emerged to enhance the benefits of heterogeneous systems by partitioning compute-intensive portions of applications (kernels) across the devices, as illustrated in Figure 1.1, where collaborative execution is employed in CPU-GPU and CPU-FPGA systems. However, rightly exploiting this environment is challenging, since each device has its particular optimization techniques that must be cooperatively employed (HUANG et al., 2019).

Figure 1.1: Collaborative execution.

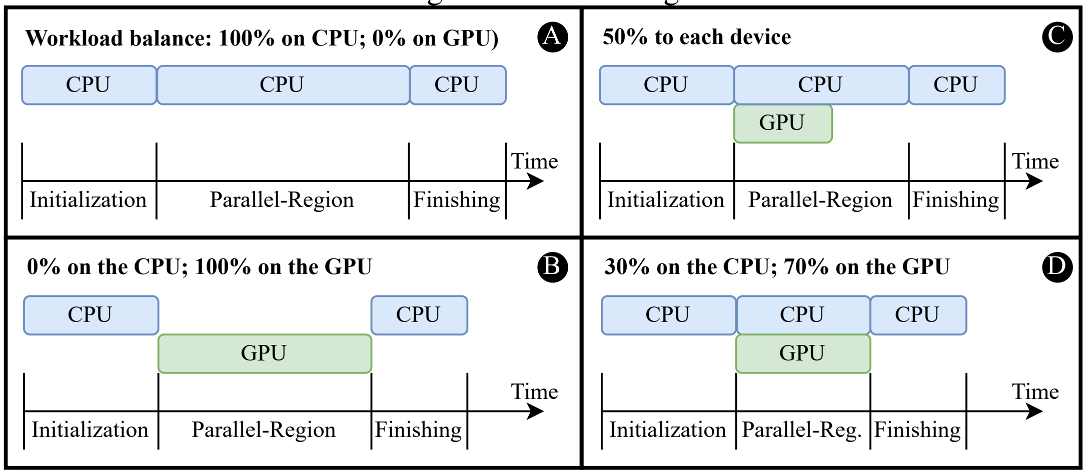

CPU-GPU Collaborative Application
CPU Kernel
CPU
GPU Kernel
GPU
CPU-FPGA Collaborative Application
CPU Kernel
CPU
FPGA Kernel
FPGA

Source: the author

1.1 Challenge

Collaborative computing presents the challenge of fully utilizing the heterogeneous devices, since these devices can vary in terms of efficiency (e.g. high-end CPU with low-end GPU/FPGA, low-end CPU with high-end GPU/FPGA), while applications present different benefits from CPUs, GPUs or FPGAs. Considering a CPU-GPU system, for instance, massive data-parallel applications are prone to have more benefits when executing on the GPU, so a workload evenly assigned between the CPU and the GPU will lead the CPU to be the bottleneck of the execution. To illustrate it, Figure 1.2 shows the execution timeline of a hypothetical application with this characteristic, and how the workload balance influences execution time. Figure 1.2-A and B present CPU-only and GPU-only executions, which do not achieve the lowest execution time possible, since the devices are not used collaboratively. Figure 1.2-C shows an even workload (in terms of code/data size to execute) balance between the CPU and the GPU. However, the GPU completes the work before the CPU, since the GPU suits best the characteristics of the data-parallel application considered. When 30% of the workload is assigned to the CPU, in Figure 1.2-D, the CPU and the GPU take similar time to execute the parallel-region, leading to a faster overall execution than evenly distributing the workload and when executing on a non-collaborative execution such as Figure 1.2-A and B.

Figure 1.2: Balancing.



Source: the author

However, collaborative applications' programmers are usually not aware of the hardware of the end-user, which prevents them from building applications capable of extracting all the benefits offered by the heterogeneous system, since specific hardware information is required to balanced the workloads as much as possible. The employment of some optimization strategies can unlock these benefits. On the CPU side, *Thread Throttling* can enable improvements to the collaborative execution. It consists in a technique that exploits TLP scalability issues by artificially adjusting the number of threads on the CPU and, consequently, varies execution time and energy consumption (LORENZON; FILHO, 2019). This way, *Thread Throttling* is capable of help balancing the collaborative execution of CPU-GPU and CPU-FPGA applications, enabling overall performance and energy improvements to these systems. High-Level Synthesis optimization techniques can further improve collaborative CPU-FPGA execution with the use of *HLS-Versioning*, which consists in applying different HLS optimizations to the design, resulting in distinct FPGA implementations for the same application. Thus, the energy-delay trade-offs presented by the different versions can offer another tuning point (ZHAO et al., 2019). A synergistic employment of the aforementioned optimizations can improve the efficiency of collaborative execution in CPU-GPU and CPU-FPGA systems.

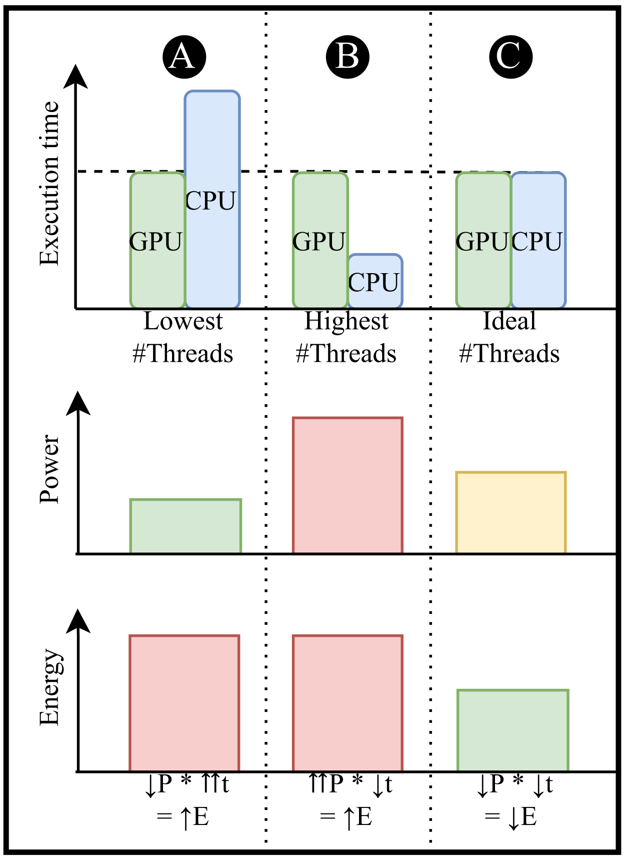
1.2 Contributions

Considering the room for improvements that collaborative environments present, this work exploits different optimization techniques in the heterogeneous devices to extend the benefits of CPU-GPU and CPU-FPGA collaborative execution. Given that, our main contributions are investigating the benefits of applying:

- CPU *Thread Throttling* in CPU-GPU data-collaborative environments, for which we propose an extension for the C++11 thread library capable of automatically employing CPU *Thread Throttling*;
- CPU *Thread Throttling* and workload balance in CPU-FPGA data-collaborative environments, for which we propose a framework that applies CPU *Thread Throttling* and selects near-optimal workload balance;
- CPU *Thread Throttling* and FPGA *HLS-Versioning* simultaneously in CPU-FPGA task-collaborative environments, in which we compared a Design Space Exploration to a heuristic approach;

Firstly, we considered the execution of data-collaborative applications over a CPU-GPU heterogeneous system and investigate how the number of threads from the multi-core CPU influences the overall performance and energy consumption of this system. Then, we evaluate the benefits of applying the CPU *Thread Throttling* technique and how the optimal number of threads is influenced by the workload balance, between the CPU and the GPU, statically defined by the programmer. To illustrate it, Figure 1.3 shows how CPU *Thread Throttling* can enhance performance and energy of a CPU-GPU collaborative execution. In Figure 1.3-A the lowest number of threads (#Threads) is used, leading the CPU to take long to execute and, as a consequence, spending a high amount of energy. The execution time is shortened when using the highest #Threads in Figure 1.3-B, but activating all CPU cores implies a rise in power dissipation, which results in high energy consumption as well since the execution is now bottlenecked by the GPU. On the other hand, when the execution is performed with an ideal #Threads (Figure 1.3-C), we keep lower power dissipation by not activating some CPU cores while execution time is still optimal. In this scenario, we propose an approach, based on the Hill-Climbing algorithm, implemented over the C++11 thread library, capable of automatically selecting a near-optimal number of threads aiming at reducing the energy-delay product (EDP) of the collaborative applications in the CPU-GPU environment.

Figure 1.3: CPU-TT on CPU-GPU collaborative execution.



Source: the author

Likewise, we investigate the influence of the CPU number of threads and the workload balance on a CPU-FPGA environment executing data-collaborative applications. Given that both fronts can offer performance and energy gains, we propose a framework capable of automatically finding a near-optimal number of threads and workload balance in CPU-FPGA applications, allowing the framework administrator to select the optimization goal between performance, energy, or EDP.

Finally, the benefits of employing optimizations at both CPU and FPGA will be investigated on the CPU-FPGA heterogeneous system. Besides using *Thread Throttling* at the CPU, *HLS-Versioning* is performed over the FPGA kernels. To evaluate the impact of both techniques, a multi-tenant Cloud service is used as an object of study, in which sequences of application requests result in execution-order dependencies between the applications. In such a scenario, it is shown that optimizing only the CPU or the FPGA side results in much fewer improvements than optimizing at both fronts. Moreover, we compare applying the optimizations to each individual kernel versus applying a single optimization for all kernels, discussing the strengths and weaknesses of each approach.

1.3 Work Structure

The remainder of this work is structured as follows.

Chapter 2 presents a background in Collaborative Computing, CPU *Thread Throttling*, and FPGA High-Level Synthesis, along with previous works in each field.

Chapter 3 presents experiments investigating the impact of the number of CPU threads and the workload balance in CPU-GPU and CPU-FPGA data-collaborative applications.

Chapter 4 presents the proposed approaches to optimize CPU-GPU and CPU-FPGA Data-Collaborative Computing environments.

Chapter 5 presents experiments investigating the impact of employing *Thread Throttling* and *HLS-Versioning* in a CPU-FPGA task-collaborative environment.

Finally, Chapter 6 draws the conclusions of this work, summarizing the results and contributions, as well as addressing possible future research.

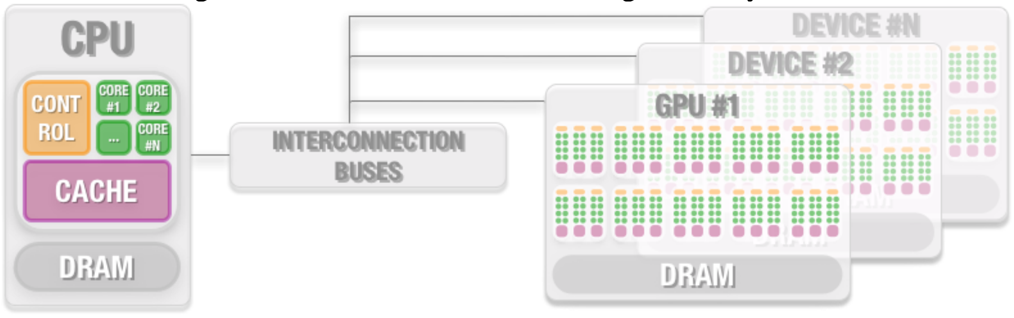
2 BACKGROUND

This chapter will provide a background on the main concepts used in this dissertation alongside the state-of-the-art of: Collaborative Computing in Section 2.1, *Thread Throttling* in Section 2.2, and FPGA High-Level Synteshis in Section 2.3. Then, Section 2.4 will show the contributions of this work over the state-of-the-art.

2.1 Collaborative Computing

Heterogeneous systems can be modeled as a set of interconnected computational resources with distributed address spaces and diverse functionalities (ILIĆ; SOUSA, 2010). Figure 2.1 shows the architecture of a heterogeneous system consisting of a host CPU along with accelerators and co-processors such as a GPU and other devices. Traditional applications offload compute-intensive parts of the application to these devices, while the CPU waits for them to finish and thus are idled, which limits the efficiency and resource utilization. In this scenario, Collaborative Computing was proposed to fill this gap by synergistically utilizing the CPU and the accelerators, and partitioning the application's workloads between the resources (ILIĆ; SOUSA, 2010).

Figure 2.1: Architecture of a heterogeneous systems.

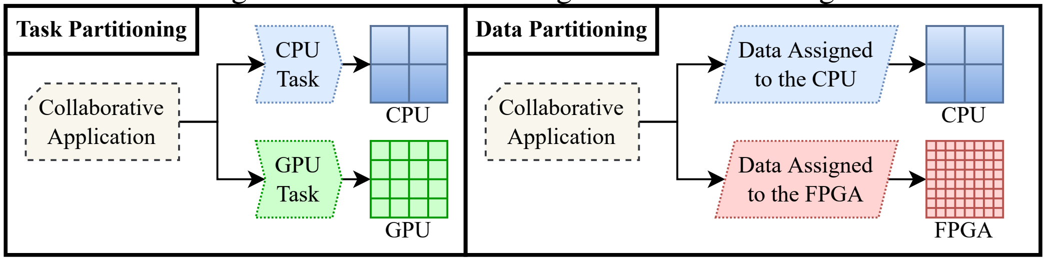


Source: (ILIĆ; SOUSA, 2010)

Properly utilizing such an environment became a challenge, though, once GPUs and FPGAs traditionally required vendor-specific programming models to use their devices. OpenCL (STONE; GOHARA; SHI, 2010) changed that by standardizing the execution in heterogeneous systems via its parallel programming standard, which abstracts most of the architectural particularities required to use heterogeneous devices so far. Heterogeneous computing in OpenCL consists of a host CPU and some OpenCL certified device, such as GPUs, FPGAs, or DSPs, for example. Its programming interface includes the support of memory transfers, allocation management, device management by the abstraction "contexts" that involve the OpenCL devices present on the system, and run-time compilation of the kernels, which are the compute-intensive portions of the program that are offloaded to the devices.

Different approaches are used to take advantage of a heterogeneous environment in a collaborative way. Figure 2.2 illustrates two of the main patterns of collaborative execution. Using Task Partitioning, the application assigns distinct parallel tasks among the devices, while using Data Partitioning the same task is performed in all devices, but the data to be computed is split between the devices. For data partitioned applications, GÓMEZ-LUNA et al. use a workload balance factor named alpha, which represents the percentage of the partitioned data assigned to the CPU, while its complementary percentage represents the data assigned to the GPU or the FPGA.

Figure 2.2: Task Partitioning vs Data Partitioning.



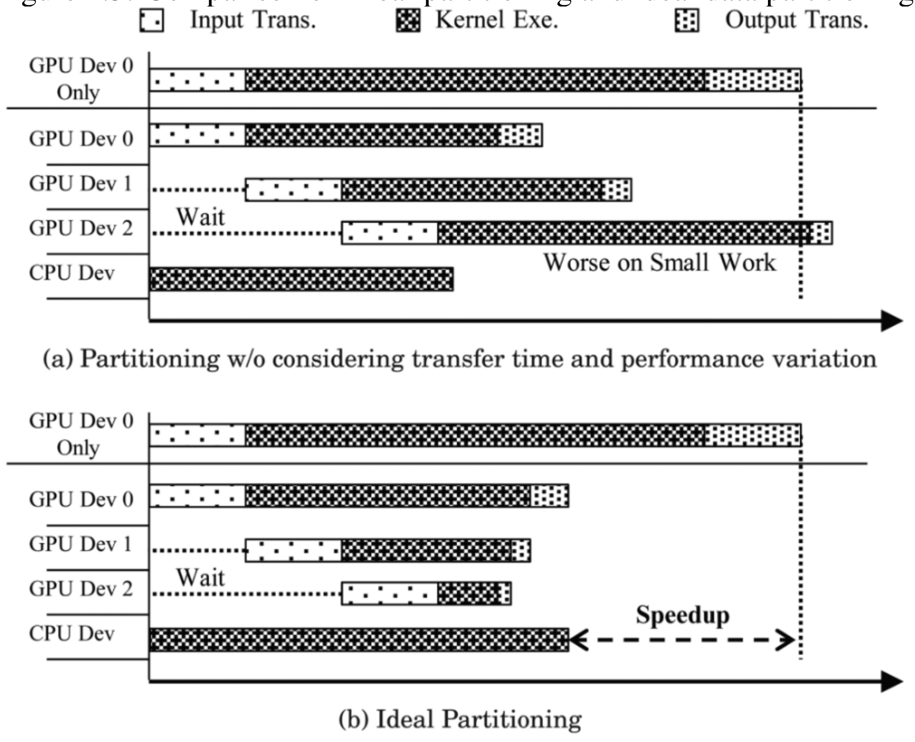
Source: the author

Several works utilize collaborative environments to propose optimizations using Task and Data Partitioning approaches. They are discussed in the following subsections.

2.1.1 Data partitioning solutions

The work proposed by LEE et al. presents the single kernel multiple devices (SKMD) system, a framework that transparently manages the collaborative execution of data-parallel OpenCL kernels across asymmetric CPUs and GPUs. It performs code transformation to enable data partition among the CPU and the GPU, evaluating transfer costs, performance evaluation, and providing a seamless result merging after the execution. Figure 2.3 from this work compares linear data partitioning among the devices (a) to the ideal partitioning (b) of a multi-GPU collaborative environment. In such an unbalanced system, the linear partition of the data among the devices (a) leads to worse performance than the GPU-only execution, while the ideal partitioning (b) provides a speedup, emphasizing the need of properly balancing the workload. The proposal achieves better performance considering a system with a multi-core CPU and two GPUs compared to the fastest device-only execution.

Figure 2.3: Comparison of linear partitioning and ideal data partitioning.



Source: (LEE et al., 2015)

The authors in (WANG; ANANTHANARAYANAN; MITRA, 2018) propose an analytical framework called OPTiC that optimizes collaborative Computing on mobile devices with thermal constraints. The framework automatically selects the workload partition between CPU and GPU by managing the OpenCL work-group assignment to the devices. It also applies frequency scaling to deliver optimal performance under mobile thermal constraints. OPTiC provides 13.68% of performance gains over existing schemes when executing CPU-GPU applications on an ARM-based platform.

CONG et al. aims at optimizing big data CPU-FPGA collaborative applications by selecting the right strategy to coordinate CPU and FPGA execution. The conducted case study uses an in-memory Samtool sort routine algorithm. They propose an adaptive dataflow execution that combines data-level parallelism on the CPU and pipeline parallelism between the CPU cores and the FPGA, making use of the CPU while offloading the computation to the FPGA, and ultimately improving overall system resource utilization. Experiments over an Intel Xeon CPU and a Xilinx UltraScale FPGA show that the proposal was capable of providing a 2.64x reduction in overall execution time.

2.1.2 Task partitioning solutions

In (ZHOU; PRASANNA, 2017) the authors conduct a study on CPU-FPGA execution of graph analytics applications. They compared Vertex-Centric and Edge-Centric paradigms, exploring the trade-offs based on their characteristics. Their proposal includes the selection of the appropriate paradigm to execute the application and a graph partitioning scheme to enable efficient parallel computation of the graphs. Experiments using two fundamental graph algorithms (Breadth-First Search and Single-Source Shortest Path) running on an Intel Xeon CPU and an Altera Arria FPGA show up to 4.2x throughput improvements over state-of-the-art FPGA-based designs.

The framework called GraphACT (ZENG; PRASANNA, 2020) addresses optimizations for neural network training, specifically Graph Convolutional Networks, on CPU-FPGA systems. The proposal integrates algorithm-architecture co-optimizations considering computation and communication characteristics of different GCN algorithms, choosing the one that is well suited for hardware execution. The framework uses a systolic array-based FPGA design for efficient parallelization, integrating load-balancing modeling where tasks are properly assigned to the CPU and the FPGA. The approach achieves speedups compared to other works while keeping low accuracy loss.

The work from MELONI et al. proposes a hardware/software solution for the acceleration of neural networks, precisely Convolutional Neural Networks, on an ARM-based SoC coupled with an FPGA. To cooperatively use the hardware, it offloads the intensive CNN workloads to the FPGA, while hard-to-accelerate tasks, which present intricate conditional statements or input/output operations, are executed on the ARM cores. The proposed accelerator uses HLS pipelining capabilities to accelerate the FPGA workloads and vector engines to speedup ARM workloads, showing 18% performance gains executing a specific CNN compared to state-of-the-art works.

DEIANA et al. also uses an ARM-FPGA SoC-based system, which proposes an approach to map and schedule applications on heterogeneous and reconfigurable devices. It presents a programming model to improve performance, power, or energy consumption by exploiting FPGA optimization techniques and task partitioning the application. For each generated task, the proposal generates multiple versions exploiting performance/energy trade-offs from FPGA optimizations. The authors evaluate the work using image and data-processing tasks, showing performance improvements in every scheduling scenario when compared to other similar work.

The work in (WEI et al., 2017) aims at optimizing the throughput of streaming applications in CPU-FPGA heterogeneous systems. It takes into account latency constraints and stringent power budgets inherent in streaming systems. The authors developed two algorithms to map tasks onto the heterogeneous system and order the application's execution considering its characteristics and architectural capabilities of the hardware. They also employed pipelining to improve the throughput by overlapping the execution of different portions of the application and using frequency scaling to adjust the execution of tasks for power saving.

2.1.3 Task and Data partitioning solutions

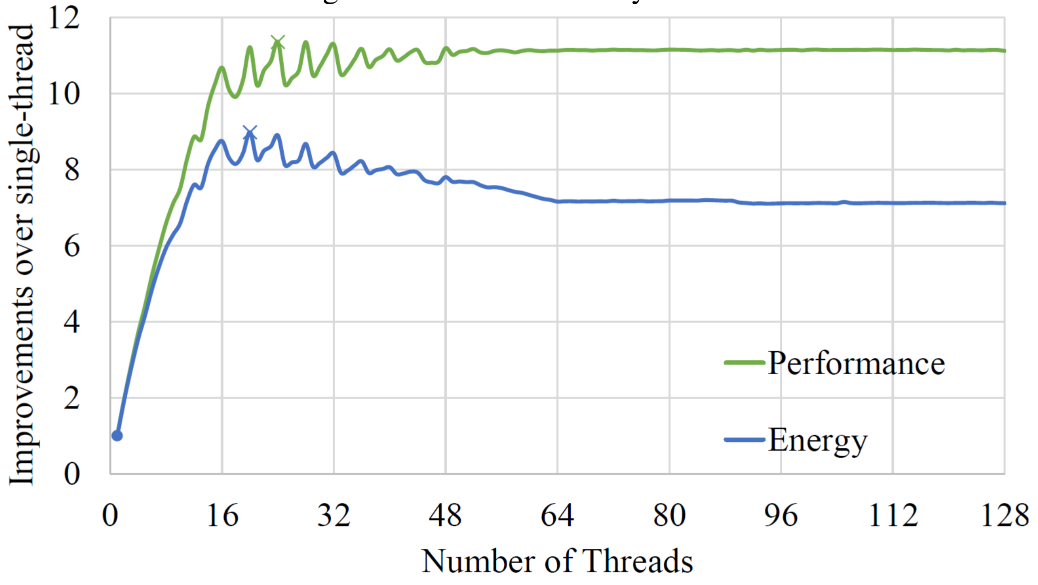
ILIĆ; SOUSA were pioneers in exploiting heterogeneous systems using a unified execution model. The authors propose an execution environment with a CPU and a GPU to execute matrix multiplication and the major of 3D FFT tasks. They used an OpenCL-alike programming model that includes the use of the Task Abstraction, which provides a seamless execution on the CPU and the GPU, a task scheduler that offers workload balancing based on an exhaustive search, as well as data partitioning capabilities. Experimental results show performance benefits when executing matrix multiplication. Contrarily, FFT had no performance benefits due to bandwidth limitations of the considered system.

(HUANG et al., 2019) was the first work to quantitatively evaluate collaborative execution with OpenCL High-Level Synthesis on CPU-FPGA systems. It brings a comprehensive analysis for task and data collaborative strategies using two CPU-FPGA systems with distinct computational power running Chai benchmarks (GÓMEZ-LUNA et al., 2017). Experimental results show that the CPU-FPGA collaborative execution outperforms conventional CPU-only and FPGA-only in all tested benchmarks. The work also provides findings on the strengths of each collaborative strategy. The use of data partitioning can enable better load balancing while keeping low communication overhead. Using task partitioning can allow more kernel duplication (a typical optimization technique that enables parallel replicated tasks to execute simultaneously in the same FPGA design).

2.2 CPU *Thread Throttling*

With the spread of CMPs, performance improvements in CPUs moved towards TLP exploitation, so that multi-threaded applications could increase concurrency using multiple simultaneous threads, ultimately reducing execution time. This strategy has limitations, though, since in several applications improvements are not linear with respect to the increase in the number of threads and, in many cases, simply using the highest number of threads possible does not provide the best performance (CURTIS-MAURY et al., 2006). Figure 2.4 shows scalability issues related to TLP exploitation in the application SRAD from Rodinia suite, performance and energy improvements (i.e. reduction in execution time and spent energy) for a 64-core (128-thread) AMD Threadripper CPU. The X-axis comprises the number of threads used to execute the application, while the Y-axis present the performance and energy improvements w.r.t the single-thread execution of the application. The most common programming approach is to use all available hardware threads - 128 threads for this processor. This experiment show that performance improvements stagnate when using more than 24 threads and energy improvements are not optimal from that point due to the increase in power dissipation caused by activating more processor cores.

Figure 2.4: TLP Scalability issues.



Source: the author

LORENZON et al. identified some of the reasons behind this behavior, such as: off-chip bus saturation, which restricts the amount of data applications can move through the cores; concurrent shared memory accesses, which are limited by the amount of memory ports offered by the hardware; and scalability issues due to application's data synchronization points. Therefore, utilizing more cores increases power dissipation, imposing energy consumption penalties when it does not bring performance improvements. Thus, artificially reducing the number of threads has proved to provide performance and energy improvements in many applications using a technique called *Thread Throttling*. Several works use this technique to optimize CPU execution.

The work from CURTIS-MAURY et al. present the challenges of achieving performance and power advantages from CMPs in multi-threaded applications. It argues that existent libraries at that time lack essential capabilities to take advantage from CMPs. Hence they propose a user-level library framework for nearly optimal online adaptation of multi-threaded code for low-power and high-performance execution. The framework is a system that dynamically changes the number of threads to improve energy efficiency. In (CURTIS-MAURY et al., 2008) the authors extend the framework to support Dynamic voltage and frequency scaling (DVFS) to enable higher energy savings.

SULEMAN; QURESHI; PATT analyzes performance limiters of multi-threaded applications. Their study found that data-synchronization and off-chip bandwidth were two main bottlenecks on improving performance. For constraint-limited applications, increasing the number of threads inflates the execution time and the power dissipation. This way, the number of threads must be carefully picked to ensure a good trade-off between performance and power. Given that, the authors propose a framework that can adapt the number of threads considering contention for locks and memory bandwidth. The framework uses CPU performance counters to collect run-time data and evaluate application behavior, using OpenMP threading library to select an appropriate number of threads for the application.

(LEE et al., 2010) is another work that applies *Thread Throttling* on multi-threaded applications. The authors claim that statically determining the number of threads of the application is very likely to be lacking for not catching dynamic conditions such as the application input set, architectural peculiarities of the hardware, and the influence of other processes on shared system resources. To address these issues, the authors present a dynamic compilation system that can automatically stitch the number of threads considering dynamic conditions.

PUSUKURI; GUPTA; BHUYAN present an approach for dynamically select the number of threads without needing source code recompilation. The authors claim that some dynamic conditions cannot be inferred at compile time, such as OS thread migrations, which can influence the optimal number of threads of the application. The proposed framework executes a given parallel application binary multiple times with a different number of threads for a short period (e.g., 100 ms), then searches for the appropriate configuration to re-run the entire application.

PORTERFIELD et al. is another work that applies *Thread Throttling* at run-time. But contrarily to the previous ones that primarily focus on reducing the execution time, this work considers energy usage. They propose an adaptive run-time system that performs automatic *Thread Throttling* based on online measurements of system power and performance data.

SHAFIK et al. propose an adaptive and scalable energy minimization model for OpenMP programs, where the programmer inserts directives in the sequential and parallel parts of the code to enable energy minimization with specified performance requirements.

MARATHE et al. propose *Conductor*, a run-time system that dynamically selects the ideal number of threads and DVFS state to improve performance under a power constraint for hybrid applications (MPI + OpenMP).

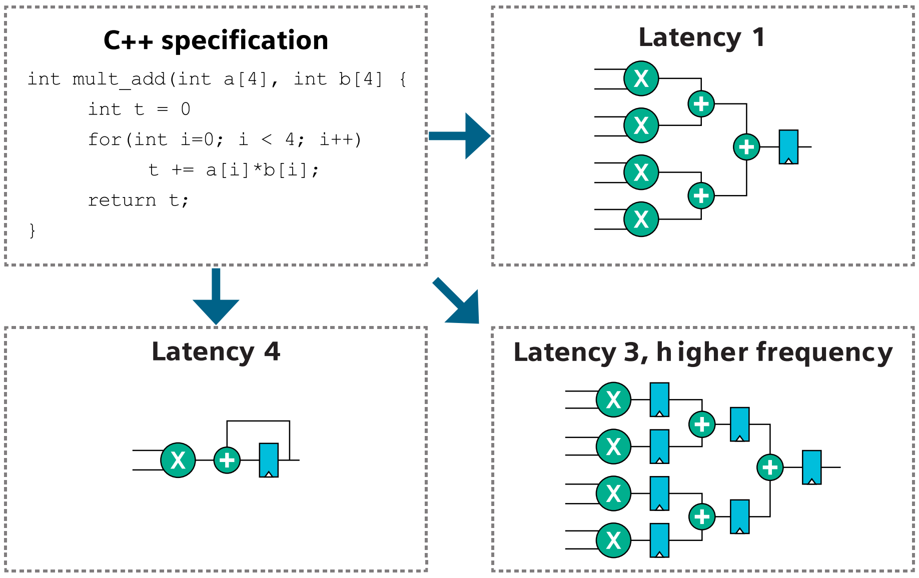
Nornir (SENSI; TORQUATI; DANELUTTO, 2016) is a run-time system that monitors the application execution and adjusts the resources configurations (DVFS, number of threads, and thread placement) in order to satisfy either performance or power dissipation requirements.

LORENZON; SOUZA; BECK propose the LAANT library that automatically adjusts the number of threads of OpenMP applications. In this approach, code must be modified by the programmer to include additional function calls in each parallel region of interest in the application. The library considers many aspects of the execution that are only possible to be defined at run-time, such as the input set of the application, the processor micro-architecture, and the distinct ideal number of threads each parallel region may have in such conditions. The approach uses a low overhead Hill-Climbing algorithm for training while the application is running to adjust the number of threads, to optimize EDP. The authors extend LAANT in the work (LORENZON et al., 2018) by adding transparency (i.e. not requiring any source-code modification or recompilation). They propose Aurora, a framework capable of finding the ideal number of threads according to a given metric defined a priori by the user. The framework achieves transparency since it relies on extending the original OpenMP library, which allows the approach to adapt at run-time the number of threads of default OpenMP applications.

2.3 FPGA High-Level Synthesis

High-level Synthesis development flow is an alternative to ease hardware development compared to traditional hardware description languages, such as VHDL and Verilog. For that, the HLS flow provides the development of designs by using high-level description languages (e.g., C/C++). The high-level description is then automatically translated to the hardware description. This approach enables easy access to various hardware optimization possibilities. Different optimizations and their combinations result in distinct versions of the same design, in the same way as distinct binaries are generated from the same source code when using different compiler flags. Figure 2.5 exemplifies how a C++ high-level source code can generate distinct hardware implementations. The multiply-accumulate operation performed by the loop can be implemented in different ways, using few hardware resources when prioritizing low power dissipation, or using more hardware when resources prioritizing low latency. Similarly, FPGAs offer different HLS optimizations such as Array Partitioning, Loop Unrolling, and Pipelining, which also present distinct variant resource consumption, processing cycles, and power dissipation, enabling design space exploration (DSE) for developers. We call this property *HLS-Versioning*. CPU-FPGA collaborative applications can also benefit from *HLS-Versioning*, once the optimal HLS version for the FPGA kernel can vary according to the application characteristics and how its workload is balanced among the CPU and the FPGA.

Figure 2.5: High-Level Synthesis.



```
int mult_add(int a[4], int b[4] {
     int t = 0
     for(int i=0; i < 4; i++)
          t += a[i]*b[i];
     return t;
}
```

Source: (TAKACH, 2016)

Finding the best combination of coarse-grained HLS optimizations can be a challenging process. The work in (PHAM et al., 2015) claims that is infeasible to explore the entire design space in *HLS-Versioning*, due to time consuming process of HLS and the exponential growth with respect to the number of design points. To address this problem the work proposes a DSE framework that exploits loop-array dependencies to reduce the evaluating time to evaluate optimal or near-optimal solutions.

ZHONG et al. propose a high-level performance estimation tool that enables rapid and accurate performance prediction for FPGA-based accelerators. Their tool named Lin-Analyzer relies on analysis techniques to avoid false data dependencies inside the high-level code and performance estimation for the FPGA HLS optimizations (loop unrolling, pipelining, and array partitioning) without generating RTL implementations. The tool performs an early DSE and assists designers in evaluation optimization configuration that best suits an application when mapped to the FPGA.

Efficiently designing applications with variable loop bounds (the number of iterations can only be known at runtime) is challenging, since typical HLS optimizations such as loop unrolling and pipelining cannot be applied. The work (CHOI; CONG, 2018) proposes a HLS-based optimization framework capable of addressing this problem through automatic code transformations that increase the utilization of computing resources by using techniques such as partial unrolling with pipelining or loop early termination.

ZHAO et al. propose a framework capable of evaluating the effects multiple directives from HLS tools and analyzing their suitability in design description. It uses pluggable analytical models, a recursive data collector, and a metric-guided DSE algorithm to perform the analysis. Given different resource constraints, the framework finds designs configurations with near-optimal performance, analyzing the trade-off relationship between performance and area. Their proposal use only a few evaluation metrics and prunes their DSE to reach the solution in a feasible time.

LIGNATI et al. propose a framework called MultiVers that uses automatic HLS generation to enhance performance and reduce energy consumption in CPU-FPGA cloud systems. It uses HLS to build libraries containing multiple versions of kernel requests in the cloud environment. The framework chooses kernel versions according to the optimization goal selected by the cloud provider (a tuple expresses a linear combination of performance, energy, and FPGA design area). According to this optimization goal, an extended version of the framework (JORDAN et al., 2021) also elects an appropriate allocation strategy for a multi-tenant cloud environment.

2.4 Contributions w.r.t. the State-of-the-Art

The studies presented in this dissertation extend the aforementioned Collaborative Computing works, which already tackled application-specific solutions. Table 2.1 summarizes the comparison of this study w.r.t. the state-of-the-art. Contrarily to TT and HLS works that focus on CPU-only and non-collaborative CPU-FPGA applications, respectively, ours apply these techniques to optimizing generic CPU-GPU and CPU-FPGA applications (not to a specific problem). Moreover, this study is orthogonal to task mapping and schedule approaches, which can be complementary used with our proposals.

Table 2.1: Comparison w.r.t. the State-of-the-Art.

Work	Thread Throttling	High-Level Synthesis	Collaborative Execution	Generic Solution
CURTIS-MAURY et al., 2006	x			x
SULEMAN et al., 2008	x			x
LEE et al., 2010	x			x
PUSUKURI et al., 2011	x			x
PORTERFIELD et al., 2013	x			x
SHAFIK et al., 2015	x			x
MARATHE et al., 2015	x			x
SENSI et al., 2015	x			x
PHAM et al., 2015		x		x
ZHONG et al., 2016		x		x
CHOI et al., 2018		x		x
ZHAO et al., 2017		x		x
LEE et al., 2015			x	x
WANG et al., 2018			x	x
CONG et al., 2015			x	
ZHOU et al., 2017			x	
ZENG et al., 2020			x	
MELONI et al., 2018			x	
DEIANA et al., 2015			x	x
WEI et al., 2017			x	
ILIC et al., 2010			x	
HUANG et al., 2019			x	x
This work	x	x	x	x

Source: the author

3 MOTIVATION TO EXPLOIT DATA-COLLABORATIVE CPU-GPU AND CPU-FPGA ENVIRONMENTS

Collaborative computing was proposed to increase the efficiency of heterogeneous systems by taking advantage of all hardware resources to perform the computation. However, as discussed in Chapter 2, extracting all the benefits of collaborative execution is challenging, since many run-time behaviors cannot be known at programming time.

This chapter investigates some of the key parameters that can impact collaborative execution, namely: the number of threads running on the CPU; and the workload balance among the heterogeneous devices. In these experiments we aim at evaluating the potential gains in terms of performance and energy obtained from applying CPU *Thread Throttling* and workload balance. Therefore, we ran a set of collaborative applications in two distinct heterogeneous systems, a CPU-GPU and a CPU-FPGA environment. To quantify the influence of each of the parameters, we explored many combinations of the number of CPU threads and workload balance factors for each of the applications.

The remainder of this Chapter is organized as follows. First, Section 3.1 introduces the methodology used in the evaluation, specifying the execution environment and Collaborative Computing benchmarks. Then, Sections 3.2 and 3.3 present the experimental evaluation investigating the optimization opportunities in CPU-GPU and CPU-FPGA collaborative environments, respectively.

3.1 Methodology

3.1.1 Execution Environment

We performed our evaluation on the system described in Table 3.1, where GPU and FPGA are connected via a PCI Express 3.0 x16, running on Ubuntu 20.04 Kernel version 5.11, NVIDIA Driver v. 460.56 and Xilinx SDAccel tool. The applications were compiled using g++ 9.3 and OpenCL Khronos ICD v. 2.2.11. We used the *Linux monitoring sensors* application to get CPU power dissipation directly from the hardware counters at run-time. The GPU power measurements were acquired using the NVIDIA System Management Interface (SMI), which provides the power draw from the whole graphics card. Since the used FPGA does not have specific power hardware counters, we acquired power using the Xilinx Vivado tool, which provides the design's power draw from the Alveo U200 FPGA, considering 25°C of ambient temperature, medium profile heat sink and 12.5% of toggle rate (switching activity). We evaluated the performance of the benchmarks by considering the time taken to execute the entire application, and evaluated energy consumption of the CPU-GPU and CPU-FPGA systems by integrating the overall application execution time and the power dissipation of all the devices from the respective systems.

Table 3.1: System specifications.

CPU / GPU / FPGA	AMD Ryzen Threadripper 3990X	NVIDIA RTX 2070 SUPER	Xilinx Alveo U200
Microarchitecture	Zen 2	Turing	UltraScale
Parallelism Avaliable	64 Cores(128 Threads)	2560 SPs	1182240 LUTs
Base Clock Frequency	2.9 GHz	1.6 GHz	Variable
Technology Node	7 nm	12 nm	16 nm
Thermal Design Power	280 W	215 W	225 W

Source: the author

3.1.2 Benchmarks

We selected Collaborative Computing benchmarks from the Chai suite (GÓMEZ-LUNA et al., 2017), which were specially developed for heterogeneous architectures, taking advantage of collaborative execution between CPU and OpenCL-compatible devices, such as a GPU or a FPGA, using the data-partitioning technique. These applications exploit CPU thread parallelism using C++11 Standard Library (ISO, 2012), which is based on C's library POSIX Threads, while exploiting GPU and FPGA parallelism through kernels implemented using the OpenCL standard (STONE; GOHARA; SHI, 2010). The benchmarks are depicted in Table 3.2 along with a brief description.

The implementation provided by this suite offers the possibility of setting many parameters from the applications via command-line arguments, such as the number of executing threads (which we vastly explore in this work), the number of repetitions for the warm-up, and timed program phases. It is also possible to specify the workload balance parameter alpha, previously described in Section 2.1. In the experiments in this Chapter, we used bash scripting to set the applications' arguments, selecting the alpha and the number of CPU threads to run each application.

Table 3.2: Chai benchmarks.

Benchmark	Description
BS	Bezier Surface
CED-D	Canny Edge Detection
HSTI	Image Histogram - Input
HSTO	Image Histogram - Output
RSC-D	Random Sample Consensus
PAD	Padding
SC	Stream Compaction

Source: the author

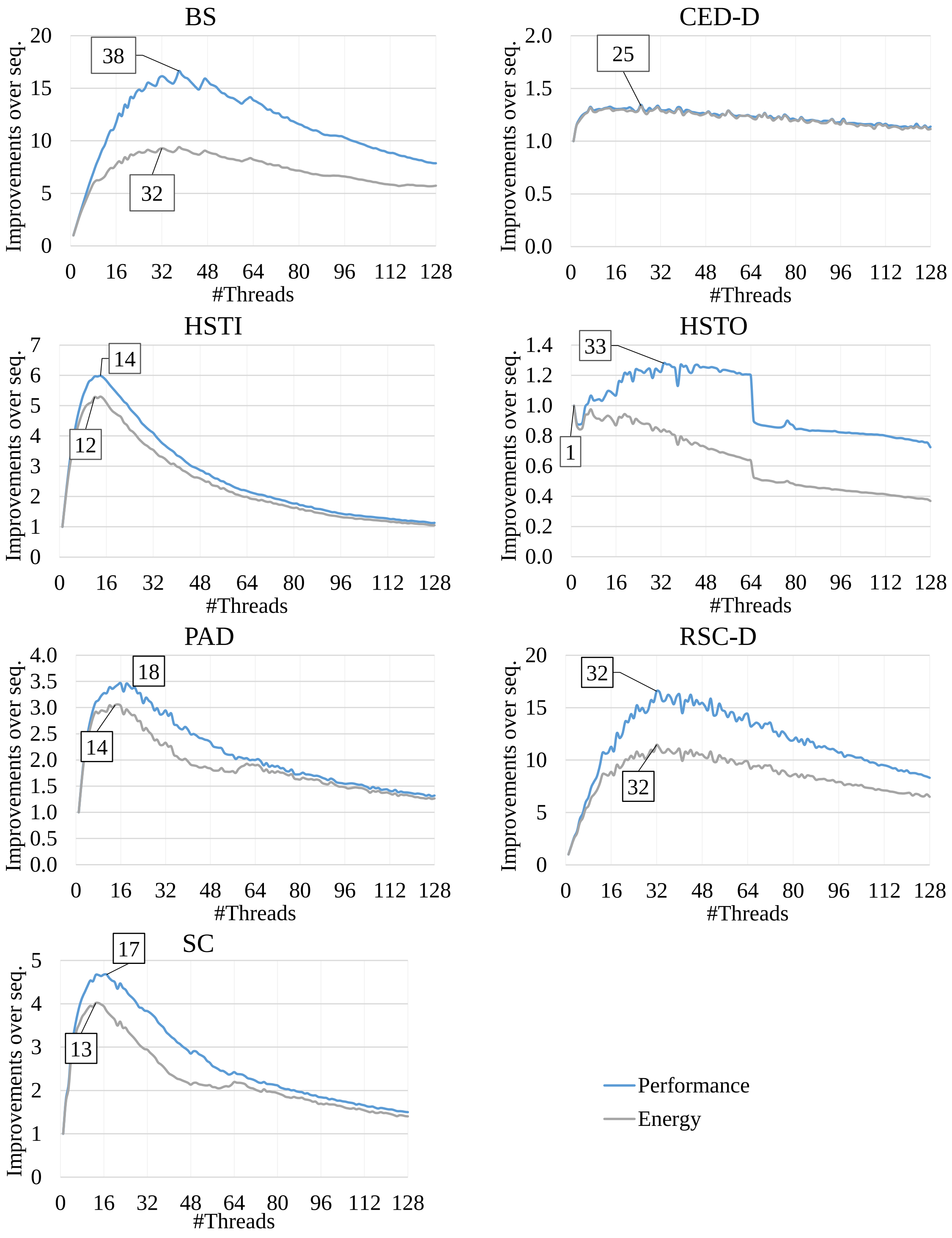
3.2 Optimization Opportunities in CPU-GPU Collaborative Environments

Intending to investigate the impact of selecting the number of CPU threads in CPU-GPU collaborative applications, we ran each of the collaborative applications from Table 3.2 using all possible CPU #Threads available in our system - from 1 to 128 threads - and using α=0.5 (i.e. half of the workload assigned to the CPU and the GPU).

Figure 3.1 shows performance (blue line) and energy (gray line) improvements, in the Y axis, when varying the CPU number of threads, in the X axis, of the CPU-GPU collaborative applications. The improvements are w.r.t single-thread execution, where values above 1.0 present a reduction in execution time (performance) and a reduction in energy consumption - 2.0x improvement means that the application is running on half of the execution time or using half of the energy, considering the collaborative environment as a whole.

As it can be noticed, in Figure 3.1, the benchmarks present variant behaviors as the number of threads in the CPU increases. HSTO, for example, achieves the greatest performance when using 33 threads, while executing with only 1 thread results in the best energy - since other #Threads result in energy degradation (values below 1.0). The increase in power dissipation caused by activating more cores does not benefit HSTO in terms of energy. Other applications such as BS, HSTI, PAD, and SC have similar behaviors but on a smaller scale, achieving optimal energy consumption using fewer threads compared to performance. Notably, none of the applications present optimal performance or energy improvements (i.e. the best efficiency) when executing with the maximum number of threads supported by the 128-threaded CPU. Once that is the most common approach in parallel applications, these experiments stand out the opportunities for improvements to the use of CPU *Thread Throttling* in collaborative applications.

Figure 3.1: CPU *Thread Throttling* Opportunities in CPU-GPU applications.



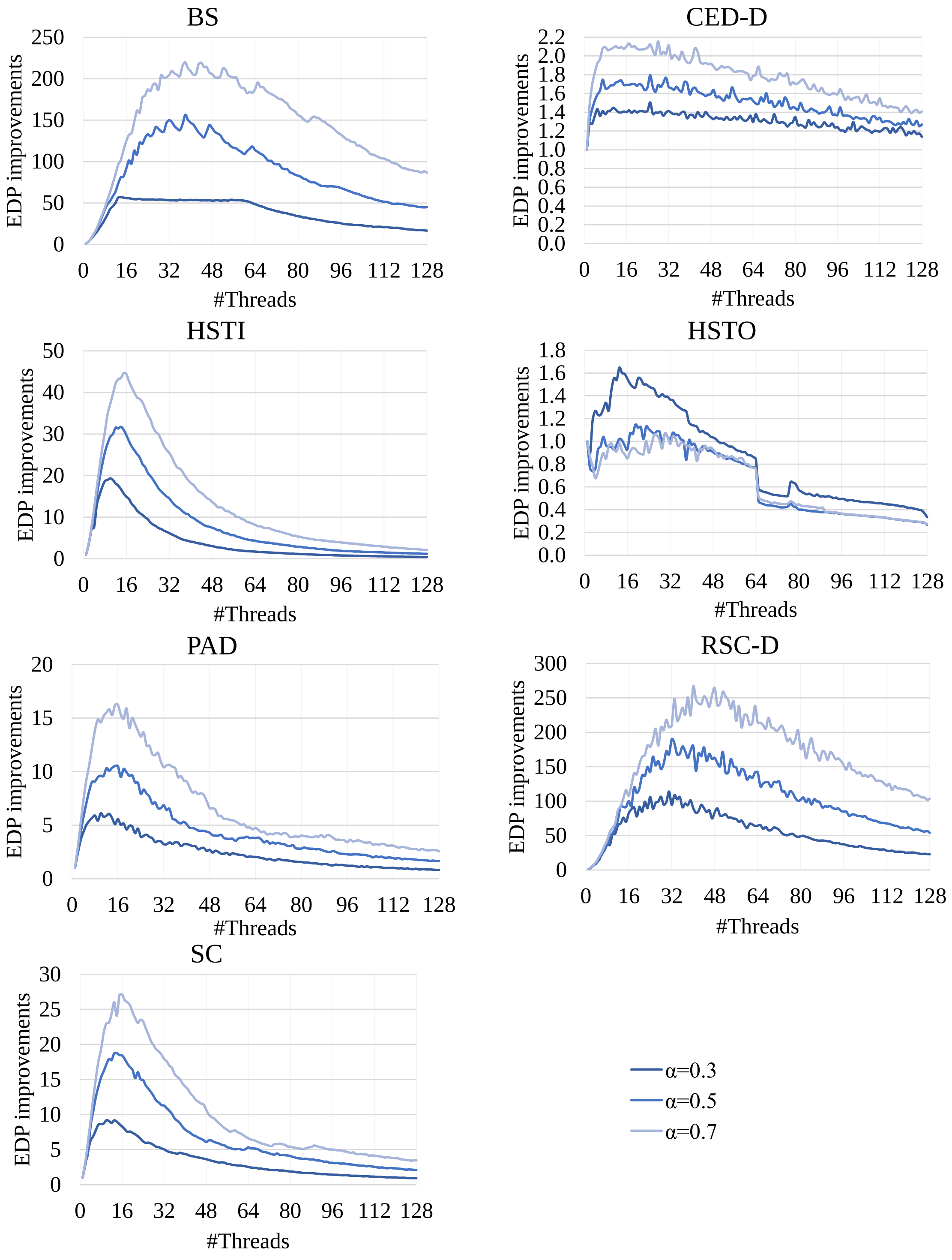
Source: the author

3.2.1 Influence of Workload Balance

This subsection presents the influence of the workload balance of the collaborative application on selecting the #Threads. We run a set of experiments evaluating the EDP improvements over the same baseline from the previous experiment - improvements w.r.t using a single CPU thread for the CPU-GPU execution. Each benchmark was evaluated with each #Thread possible using different workload balances, with three alpha levels (0.3, 0.5, and 0.7). The experiments are normalized w.r.t. to the execution using one CPU thread of the respective alpha being evaluated (e.g. EDP curves for α=0.3 were drawn using 128 threads with α=0.3, and so on).

One can observe in Figure 3.2 that most benchmarks present different EDP regarding the combination of #Threads and alpha. BS and HSTO, for instance, show optimal EDP with distinct #Threads when alpha varies. BS has optimal EDP using 14 CPU threads when α=0.3 and 38 threads when α=0.7; while HSTO shows optimal EDP with 13 and 30 threads in the same scenarios. These behaviors occur because in scenarios where more data has to be computed by the GPU (α=0.3), the CPU naturally has a lower demand and do not need to execute its workload as fast as possible. So when using a high #Threads the CPU will finish its parallel-region earlier than the GPU and remain idle until the GPU completes its processing. Instead, a lower #Threads can keep the same performance with a lower energy consumption, since less CPU cores are activated, which ultimately improves the EDP. In summarizing, a greater #Threads is more efficient when more work is assigned to the CPU and a fewer #Threads when the GPU has a greater workload.

Figure 3.2: Influence of Workload Balance in CPU-GPU applications.



Source: the author

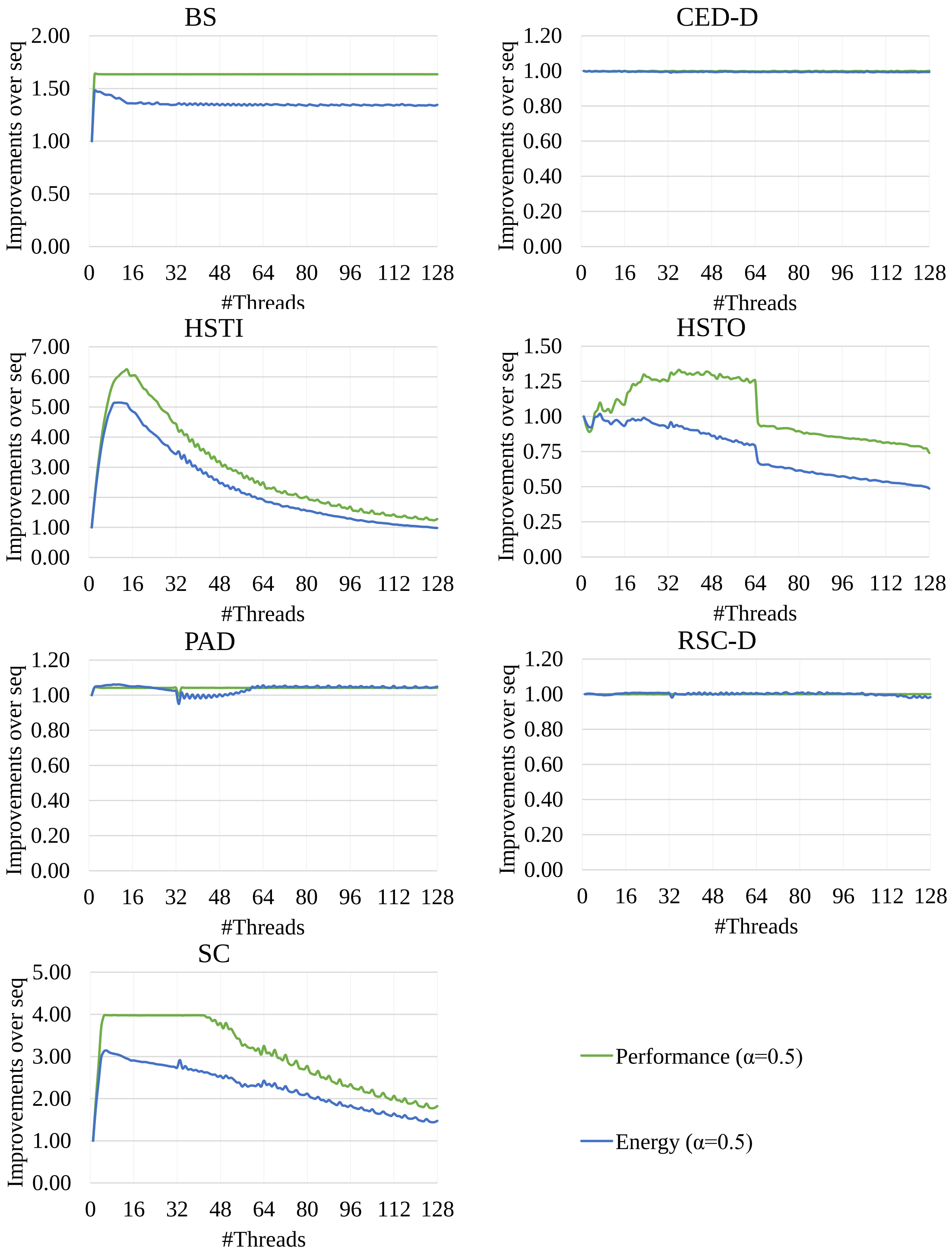
3.3 Optimization Opportunities in CPU-FPGA Collaborative Environments

This section investigates the impact of selecting the number of CPU threads in CPU-FPGA collaborative applications, as well as analyzing the influence of the workload balance to optimize the collaborative execution, using the same benchmarks and execution environment previously used in the CPU-GPU applications. This way, we ran each collaborative application from Table 3.2 using all possible CPU #Threads available in our system - from 1 to 128 threads - using 50% of workload balance factor (α=0.5).

Figure 3.3 shows in the Y-axis the performance (green line) and energy improvements (blue line) in a CPU-FPGA environment when varying the number of CPU threads of the application (X-Axis) - improvements w.r.t the execution over a single CPU thread using 50% workload balance ($\alpha = 0.5$) between CPU and FPGA.

As it can be noticed, the applications present variant behaviors as we vary the number of threads in the CPU and the evaluated metric. On the CPU side, the execution with the maximum possible number supported by the 128 threaded CPU does not present optimal performance or energy improvements (i.e., the best efficiency) for many applications, such as HSTI, HSTO, and SC. Also, in these applications the optimal number of threads is not the same for performance and energy. In BS, CED-D, PAD, and RSC-D varying the #Threads does not influence performance and energy, since they present and unbalanced execution using $\alpha = 0.5$. This behavior occurs because this benchmarks have greater performance on the CPU side, and thus the execution is bottlenecked by the FPGA.

Figure 3.3: CPU *Thread Throttling* Opportunities in CPU-FPGA applications.



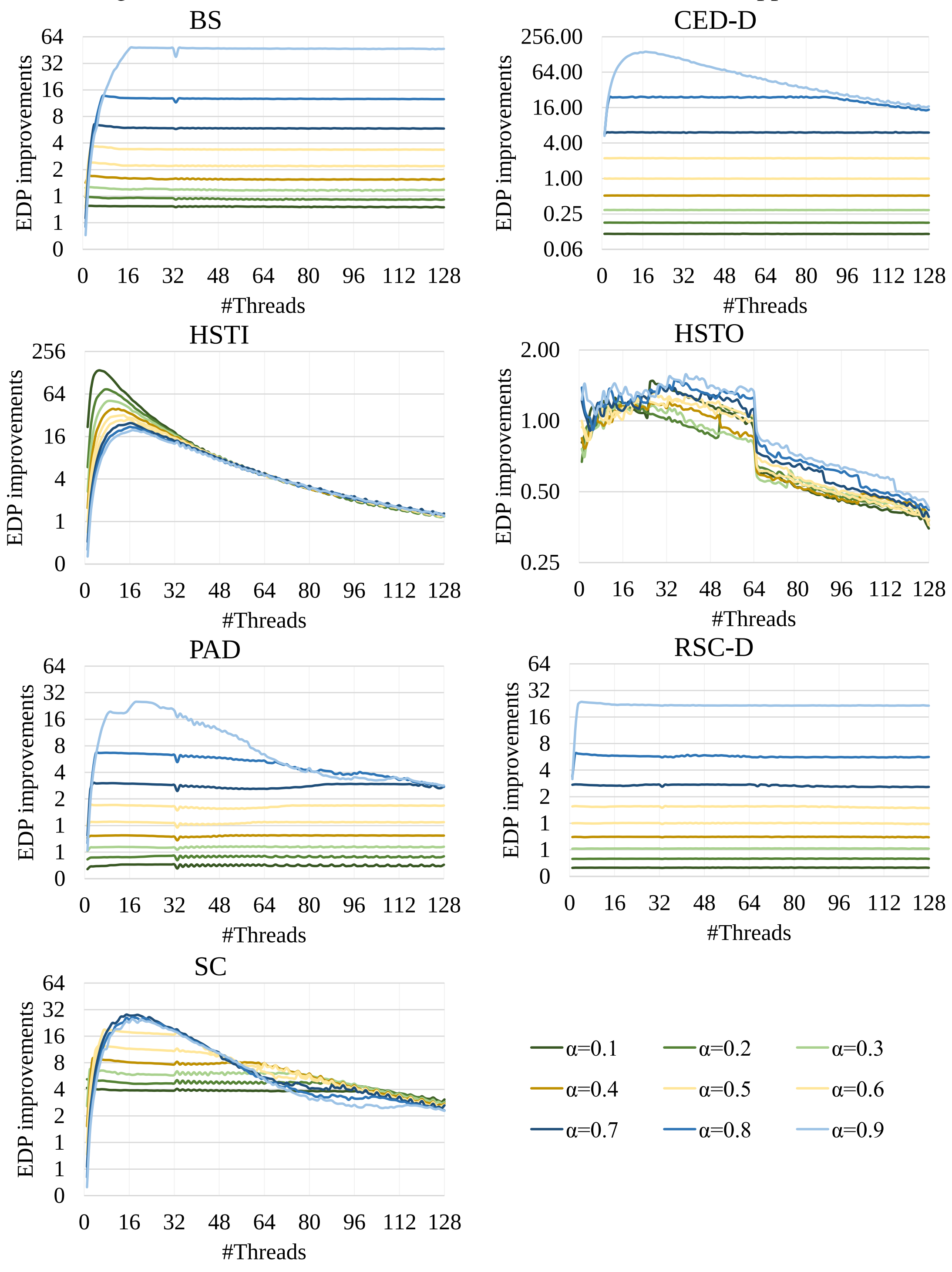
Source: the author

3.3.1 Influence of Workload Balance

Extending the exploration around the influence of workload balance, this section presents the impact of varying the workload balance from α=0.1 to α=0.9 with steps of 0.1, while changing #Threads from 1 to 128. Figure 3.4 present the EDP of each application normalized w.r.t. using 128 CPU threads and α=0.5.

At first glance, it is possible to notice that each application has different behavior. HSTI presents an ideal EDP with α=0.1 and only #Threads=4, so it shows more benefits when executing most of the kernel's data in the FPGA. When using many threads, especially more than 64, EDP is negatively impacted, because Simultaneous Multithreading (SMT) is activated to execute more than 1 thread per core, which does not provide performance benefits in this case. In contrast, BS, CED-D, PAD, and RSC-D have similar EDP regardless of the #Threads, but have huge improvements with the changing of the workload balance. The more data is computed on the CPU, the more benefits in EDP are achieved since the ideal EDP occurs when 90% of the data is computed by the CPU. Lastly, HSTO and SC show a mixed behavior, where both workload balance and #Threads influence EDP. Single-threaded execution of SC with α=0.1 provides better EDP than using all hardware threads available and α=0.9, which may indicate that FPGA suits best for this application. However, the ideal EDP occurs when executing with 21 threads and α=0.9, which clearly shows the need for tuning both #Threads and workload balance in a collaborative scenario to minimize EDP.

Figure 3.4: Influence of Workload Balance in CPU-FPGA applications.



Source: the author

3.4 Discussion

The experiments presented in this Chapter showed how the number of CPU threads impacts performance and energy in CPU-GPU and CPU-FPGA collaborative execution. Also, for both systems, the optimal #Threads varies according to: the characteristics of the application; the evaluated metric (performance/energy/EDP); and the workload balance among the devices. Therefore, the challenge is twofold: it lies in finding the optimal number of threads and workload balance; which vary according to the preferred metric.

Facing these challenges, in Chapter 4 we propose approaches to optimize the collaborative execution by synergistically selecting the number of CPU threads while considering the workload balance of the application. Since many of the CPU-FPGA applications have big performance/energy impact by the workload balance, our proposal for this environment also adjusts the alpha parameter. Is important to point out that at this point of the dissertation, we still do not optimize the FPGA side of the collaborative execution using HLS optimizations, which will be considered in Chapter 5.

4 OPTIMIZATION OF DATA-COLLABORATIVE COMPUTING EXECUTION USING CPU *THREAD THROTTLING*

This chapter presents the studies developed in this dissertation around approaches to optimize Collaborative Computing environments. Precisely, we propose two optimization approaches using CPU *Thread Throttling* considering the workload balance among the heterogeneous system.

Section 4.1 proposes Energy-Aware CPU *Thread Throttling* Approach for CPU-GPU Collaborative Environments (ETCG), an extension for the C++11 thread library capable of automatically employ CPU *Thread Throttling* on CPU-GPU collaborative applications. It can be applied to any parallel application developed using C++11 thread library by annotating code in the parallel-regions. It uses an approach based on the Hill-Climbing algorithm to select a near-optimal number of threads aiming at minimizing the EDP of an application in a CPU-GPU collaborative environment.

Section 4.2 proposes Energy-Aware *Thread Throttling* and Workload Balancing Framework for CPU-FPGA Collaborative Environments (ETCF), which comprises a framework that extends ETCG's *Thread Throttling* capabilities to synergistically determine a near-optimal number of CPU threads and workload balance of CPU-FPGA collaborative applications (i.e., determining a near-optimal combination of #Threads and alpha). To perform *Thread Throttling*, ETCF uses the same C++11 thread library extension proposed for ETCG, while using OpenCL to perform workload balancing. It also offers to the framework administrator three optimization goals: performance; energy; and EDP, which guide ETCF's algorithm when selecting the #Threads and workload balance.

4.1 ETCG: Energy-Aware CPU *Thread Throttling* Approach for CPU-GPU Collaborative Environments

4.1.1 ETCG Overview

The ETCG is a function set implemented inside the C++ Thread Library that enables transparent and run-time selection of a near-optimal number of CPU threads for CPU-GPU architectures. ETCG aims at minimizing the EDP, balancing the trade-off between performance and energy consumption, considering the performance of the entire

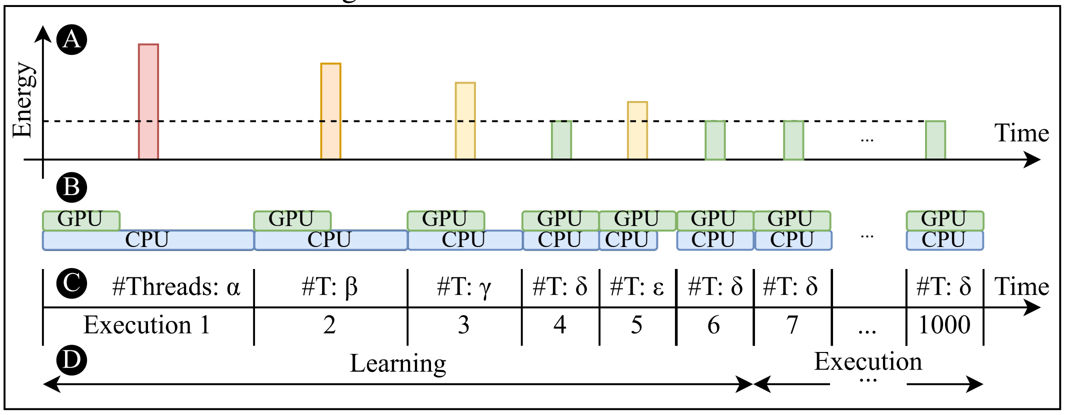
CPU-GPU collaborative execution and the energy consumption of both devices. Despite focusing on EDP, ETCG can be easily adapted to focus on performance or energy, individually.

4.1.1.1 Execution Flow

Before detailing how ETCG performs the search for the near-optimal number of CPU threads, we will explain in a higher level its execution flow, illustrated in Figure 4.1. In this Figure, a collaborative execution is performed over a parallel region. Figure 4.1-C shows the parallel region execution time, which is comprised a loop with 1000 iterations in this example. The energy consumption of each iteration is depicted in Figure 4.1-A, while Figure 4.1-B represents CPU and GPU proportion in the execution time of each iteration, and, finally, Figure 4.1-D shows the two main ETCG phases: Learning Phase and Execution Phase.

The Learning phase represents the iterations where ETCG is searching for the ideal CPU #Threads, while the Execution Phase runs the rest of the application using the number of threads found by ETCG (Figure 4.1-D). As we only vary the number of threads in the CPU, only the CPU execution is affected by ETCG (Figure 4.1-B). As it can be noticed in Figure 4.1-A and Figure 4.1-C, the first iterations of the parallel region take more energy and time to execute since ETCG is converging to its near-optimal solution. In the example, our approach takes only 7 iterations to converge to a near-optimal number of threads.

Figure 4.1: ETCG execution flow.



Source: the author

It is essential to notice that a run-time approach needs to converge to its solution in a feasible time since iterations using a #Threads far from the optimal degrade the application's EDP. Thus, an exhaustive search is prohibitive since: 1) it tests all the number of threads, even those that present high EDP degradation; and, 2) its convergence time may be longer than the parallel region execution or represent a considerable portion of its total execution time, depending on the number of threads supported by the CPU and the number parallel region iterations. Meanwhile, ETCG is suitable for run-time scenarios, once its approach finds a near-optimal solution in few iterations.

Selecting the number of CPU threads when programming an application is also not trivial, since the each one have variant behaviors according to their suitability to the CPU and GPU, as previously shown in Figure 3.1. Given that, our approach takes advantage of optimizing the application considering the run-time performance and energy measurements gathered during its Learning Phase. This phase comprises a Hill-Climbing based algorithm searching for a near-optimal number of CPU threads, implemented in a function inside our C++11 thread library extension, which collects the run-time performance and energy presented using different #Threads in the initial iterations of the parallel-region. Next, we detail how the programmer can use the ETCG's set of functions and how the search algorithm was implemented.

4.1.1.2 Search Algorithm

ETCG uses a Hill-Climbing-based heuristic to adjust the number of CPU threads, which has already proven to be efficient by LORENZON et al., illustrated by Algorithm 1. The algorithm is divided into two phases: *Initial* and *Search*. The *Initial* phase serves as a baseline to guide the Hill-Climbing algorithm, establishing the best starting point of the search. The *Search* phase uses the Hill-Climbing logic to find the number of threads that present the minimum EDP.

The *Initial* phase starts executing the parallel region (e.g. loop iteration) using the maximum number of threads supported by the hardware *(line 1)*. Then, it runs with half of this number *(line 2)*. For both executions, it collects energy and delay measurements to evaluate their resulting EDP.

Then *Search* phase starts using the measurements gathered during the *Initial* phase to start the Hill-Climbing algorithm. Summarizing, it compares the EDP of the execution considering the maximum and half number of threads. It chooses the one with lower EDP, which defines the initial range of search, then it continues searching inside this range.

To converge to the ideal number of threads, *Search* phase compares the EDP obtained by executing the parallel region considering the current number of threads (current-

Measure) and the EDP obtained in the last iteration (lastMeasure). *If there is a reduction in EDP (line 10)*, it checks if the algorithm converges up-hill (increase in the number of threads – *line 13*) or down-hill (decrease in the number of threads – *line 11*) and continues with the convergence. In other words, if the algorithm is in the up-hill direction, it continues increasing the #Threads (*line 14*). Else, it decreases the #Threads (*line 12*). Otherwise, *if there is an increase in EDP (line 15)*, it evaluates the algorithm direction and goes to the opposite direction. In Algorithm 1, *step* denotes the adjustment rate, which reduces at each iteration of the algorithm until the end. The algorithm's loop repeats its procedure cutting *step* by half each iteration, while the *step* is higher than one. When step reaches one, the algorithm returns the number of threads found.

Algorithm 1 Search Algorithm

```
 1: Initial phase:
 2:     $lastNT \leftarrow maxThreads$
 3:     $currentNT \leftarrow halfMaxThreads$
 4:     $lastMeasure \leftarrow$ ExecuteNmeasure($lastNT$)
 5:     $currentMeasure \leftarrow$ ExecuteNmeasure($lastNT$)
 6:     $step \leftarrow halfMaxThreads \div 2$
 7:
 8: Search phase (Hill-Climbing):
 9:     while $step > 1$ do
10:         if $currentMeasure < lastMeasure$
11:             if $currentNT < lastNT$
12:                 $nextNT \leftarrow currentNT - step$
13:             else
14:                 $nextNT \leftarrow currentNT + step$
15:         else
16:             if $currentNT < lastNT$
17:                 $nextNT \leftarrow currentNT + step$
18:             else
19:                 $nextNT \leftarrow currentNT - step$
20:
21:         $lastNT \leftarrow currentNT$
22:         $currentNT \leftarrow nextNT$
23:
24:         $lastMeasure \leftarrow currentMeasure$
25:         $currentMeasure \leftarrow$ ExecuteNmeasure($currentNT$)
26:
27:         $step \leftarrow step \div 2$
28:
29:     return $currentNT$
```

Our implementation of the Hill-Climbing uses the half-interval search method to reduce convergence time, meaning that the step length is reduced by half each iteration. This way, the number of comparisons is O(log2 n), where n is the number of threads supported by the hardware, so a 128 threaded CPU would take only 7 executions to converge to the result plus the *Initial* execution. Next, we show the ETCG integration to the C++ standard library and how ETCG can be applied to applications.

4.1.1.3 Integration to C++ library

C++11 Standard Library (ISO, 2012) inserted thread-management execution support inside C++ standard, so any application written in C++ could benefit from TLP exploitation without relying on external libraries. ETCG was incorporated into this library so that the developer can use our approach by calling the implemented set of functions into his source code. In this way, the program can achieve near-optimal execution, independently from the hardware (CPU and GPU) at hand. We adapted the OpenMP approach from LORENZON et al. to our C++ thread library extension, which provides rapid convergence on searching for a near-optimal number of CPU threads.

The library extension is composed of three functions that implement the algorithm. Function *initETCG* represents the Initial phase, while *startParallelRegion* and *endParallelRegion* do the Search phase of the algorithm.

Next, we describe each function of the library in details:

- initETCG() function initializes variables and structures used to control the Hill-Climbing algorithm. It also initializes the libraries used to collect information from parallel regions' behavior, such as the execution time and energy measurements.
- startParallelRegion() set the #Threads and initializes time and energy counters to measure these metrics under the given #Threads.
- endParallelRegion() captures time and energy counters to evaluate metrics and run an iteration of the Algorithm 1, adjusting the #Threads for the next iteration.

ETCG functions use the *chrono* library to measure execution time, which is part of the C++ standard and therefore linked in compile time. Likewise, energy consumption measurements are obtained directly from hardware counters present in modern processors. Intel counter statuses can be read using the Running Average Power Limit (RAPL) library (HÄHNEL et al., 2012), while the Application Power Management library can be used to read the counters of AMD processors (HACKENBERG et al., 2013).

4.1.2 Methodology

We evaluated ETCG using the same benchmarks from Chapter 3, and the CPU and the GPU specified in Table 3.1, running upon the same OS and drivers, as well as using the same compiler and tools to collect power data.

4.1.2.1 Compared scenarios

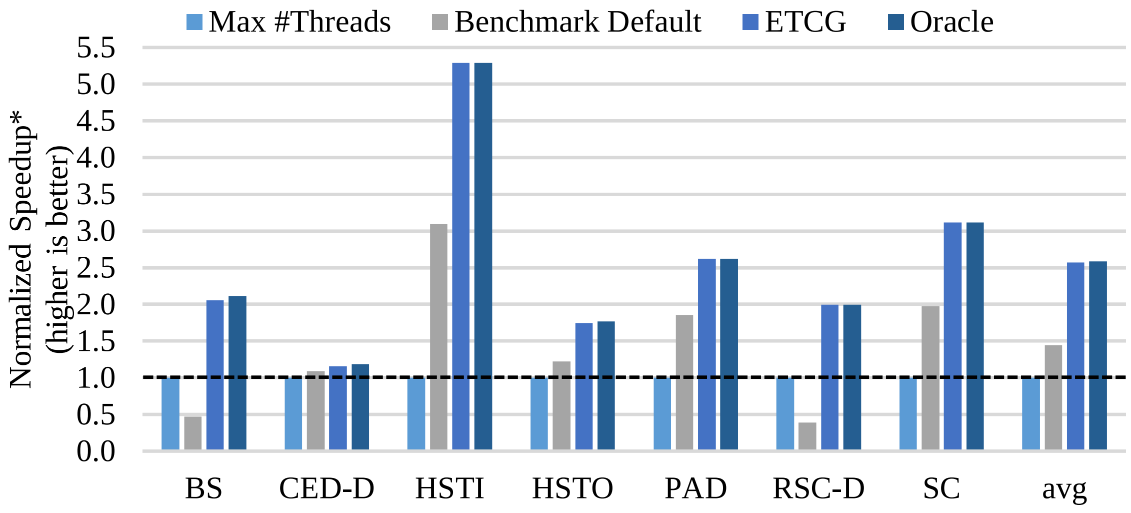
We used three scenarios to validate the proposal according to the TLP exploitation when executing the benchmarks:

- Max #Threads: Used as the baseline, where #Threads is set to the maximum number of supported threads that can be executed concurrently according to the hardware available in the system (in our case, 64 cores with SMT can run 128 threads in parallel). The max #Threads is used by default by many other parallel APIs, such as OpenMP;
- Benchmark Default, where we do not provide the #Threads when launching the program. Therefore, it runs with the default #Threads (4 threads), as programmed in the original benchmarks without any modifications (and therefore how the regular user would run these applications);
- ETCG, where the #Threads is defined after applying our Hill-Climbing based algorithm as described in Section III;
- Oracle, which represents the optimal #Threads gathered offline from an exhaustive search.

We evaluated ETCG in each scenario considering three different metrics: performance, energy consumption, and EDP. For each scenario, we kept the workload balance factor with α=0.5 (i.e., half of the data assigned to each device). Additionally, we examined the overhead imposed by the proposed search algorithm by comparing the convergence time of ETCG with the exhaustive search.

4.1.3 Results

Figure 4.2: Performance improvements normalized w.r.t the baseline *Max #Threads* (α=0.5).



Source: the author

Figure 4.2 shows the performance results considering the scenarios, described in Section 4.1.2.1, normalized w.r.t. the *Max #Threads* scenario, which uses α=0.5. Compared to the *Max #Threads*, our approach (*ETCG*) provides an average speedup of 2.56x. The reason why performance improves when the number of threads decreases is that many times the thread creation overhead is very significant. This overhead makes the CPU take longer to run its portion of kernels' data, while the GPU remains idle while waiting for the CPU data to be processed. For the same reason, the *Benchmark Default* scenario also shows performance improvements when compared to the *Max #Threads* scenario, since some applications have low thread scalability (LORENZON et al., 2018).

We can observe that *ETCG* provides similar improvements compared to the *Oracle* approach, with an average speedup of 2.56x, almost the same as the speedup obtained by the *Oracle* (2.58x). BS shows the most significant disparity compared to *Oracle*, where *ETCG* shows a 6% smaller speedup. Hence, ETCG is capable of providing near-optimal performance with negligible search overhead.

Table 4.1 shows the #Threads provided by *ETCG* and *Oracle*'s exhaustive search. It is noticeable in some benchmarks, as HSTI, RSC-D, and SC, both approaches provide the same #Threads, but in most cases they diverge, once *ETCG* takes only a few search steps to provide its solution and eventually fall into local minima (JOHNSON; JACOBSON, 2002). Despite that, its solutions are capable of reaching near-optimal results.

Table 4.1: #Threads found by Oracle and ETCG searches.

Benchmark	BS	CED-D	HSTI	HSTO	PAD	RSC-D	SC
Oracle	38	25	14	19	15	32	13
ETCG	32	14	14	28	16	32	13

Source: the author

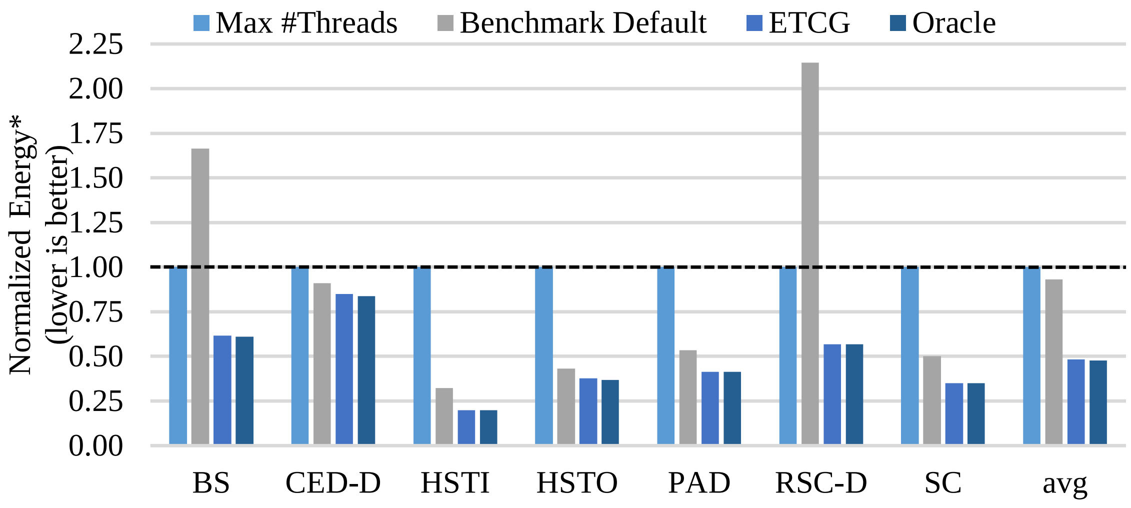
Table 4.2 shows the time taken to reach the solutions from Oracle and ETCG. The Oracle's exhaustive search suffers from a combinatorial explosion when searching for the optimal #Threads, which results in a huge time to solution. Instead, ETCG is capable of providing near-optimal #Threads in up to 6% of the time in the worst case and only 4.8% on average. Hence, exhaustively searching for the solution proves to be unfeasible once time to solution is crucial for a run-time approach such as ETCG.

Table 4.2: Time to solution (ms).

Benchmark	Total Execution Time	Oracle Search Time	ETCG Search Time	ETCG/Oracle Fraction
BS	2676	593	27	4.5%
CED-D	5335	749	45	6.0%
HSTI	9942	3729	147	3.9%
HSTO	7394	1258	68	5.4%
RSC-D	2855	654	31	4.7%
PAD	14310	3120	142	4.5%
SC	2186	560	25	4.5%
Average	6385	1523	69	4.8%

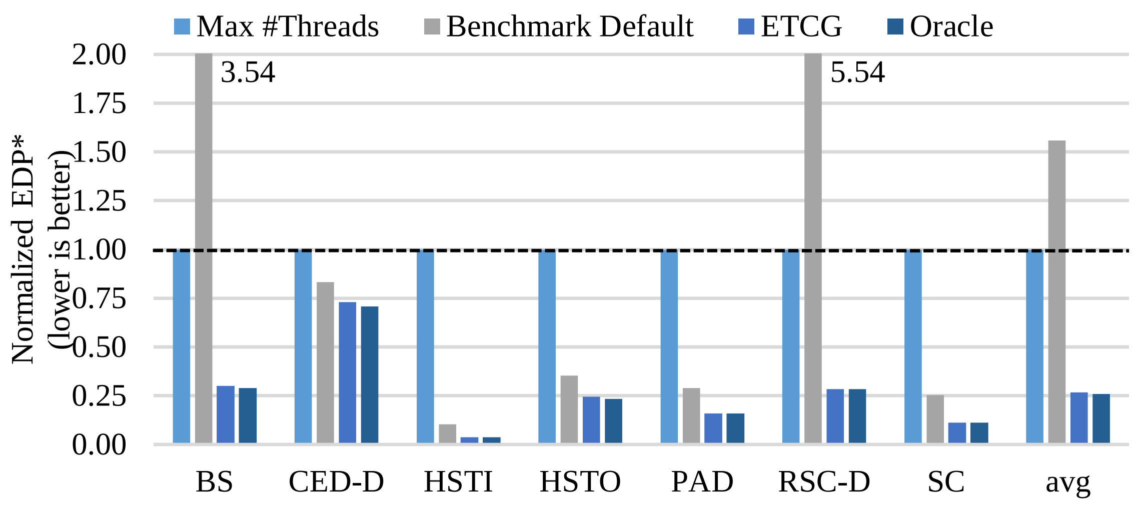
Source: the author

Figure 4.3: Energy consumption normalized w.r.t the baseline *Max #Threads* (α=0.5).



Source: the author

Figure 4.4: EDP normalized w.r.t the baseline *Max #Threads* (α=0.5).



Source: the author

Figure 4.3 and Figure 4.4 show the energy and EDP results compared to the *Max #Threads* baseline (α=0.5). Figure 4.3 shows that the *Thread Throttling* provided by ETCG reduces energy consumption compared to the *Max #Threads*. As previously shown in Figure 4.2, *Max #Threads* does not provide the best performance, which reflects in energy results in Figure 4.3, since it takes longer to execute while increasing power dissipation. Considering that fact, since the *Max #Threads* execution keeps all CPU cores active, it degrades overall energy consumption when performance improvements are small (the trade-off between extra performance and power dissipation does not pay off). In the same way, the *Benchmark Default* #Threads scenario is capable of providing a better energy efficiency than executing with the maximum #Threads, even though not presenting as good results as *ETCG*.

Summarizing, *ETCG*'s heuristic is capable of automatically reaching similar #Threads to *Oracle*, as previously mentioned in Table 4.2. CED-D and HSTO are examples where it does not occur, since they present a plateau in EDP results for a range of #Threads (CED-D between 14 and 32 #Threads and HSTO from 13 to 30). Still, *ETCG* provides solutions with less than 3% of EDP divergence compared to the *Oracle* for all applications. In the end, *ETCG* provides a dynamic and flexible solution capable of achieving near-optimal results for performance and energy consumption metrics. Appendix A presents tables with the experiments collected for the ETCG evaluation.

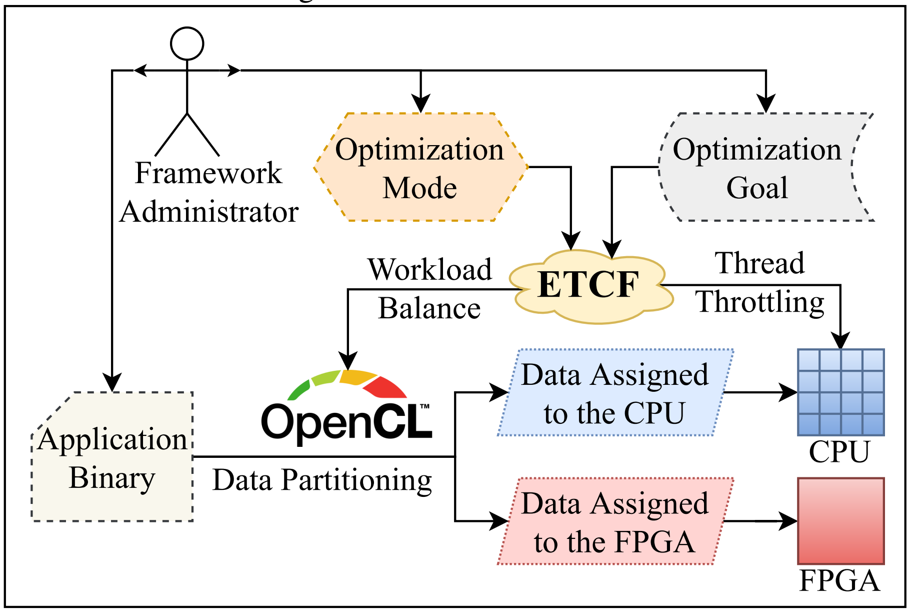
4.2 ETCF: Energy-Aware *Thread Throttling* and Workload Balancing Framework for CPU-FPGA Collaborative Environments

4.2.1 ETCF Overview

The ETCF is an approach that improves the efficiency of CPU-FPGA execution. ETCF extends ETCG, in which we explored *Thread Throttling*, to support the synergistic selection of the workload balance (i.e. varying the alpha parameter of data-collaborative applications). To achieve that, the framework offers three optimization goals to the administrator: performance, energy, or EDP. Figure 4.5 depicts the ETCF framework. Given an optimization goal, the framework is capable of detecting the near-optimal workload balance and near-optimal number of threads to a given C++/OpenCL application.

The ETCF framework offers three optimization modes: ETCF Workload Balance (ETCF-WB), ETCF *Thread Throttling* (ETCF-TT), and Full ETCF, which uses both techniques (ETCF-WB + ETCF-TT). The framework administrator is responsible for choosing one of these modes, as well as providing the optimization goal and the application binary, as Figure 4.6 illustrates. Figure 4.6-A illustrates the ETCF-WB mode, which is an offline approach that searches for a near-optimal workload balance of an application's data-parallel tasks, enabling the framework administrator to select the optimization goal (i.e., performance, energy or EDP). This mode receives as input the optimization goal and the OpenCL application. Figure 4.6-B shows ETCF-TT, an online approach that automatically selects a near-optimal number of CPU threads. It requires the insertion of ETCF C++ functions into the code, which collects runtime data and adjusts the number of threads according to the defined optimization goal. Figure 4.6-C illustrates Full ETCF mode, which comprises both ETCF-WB and ETCF-TT. It can synergistically search for the ideal workload balance and the number of CPU threads. As will be presented in the results, this approach is capable of improving the gains when compared to the decoupled use of ETCF-WB and ETCF-TT. This mode receives as input the optimization goal and the C++/OpenCL application implemented inserting our ETCF-TT C++ functions. In the next subsections, we discuss in detail all the aforementioned modes.

Figure 4.5: ETCF Overview.



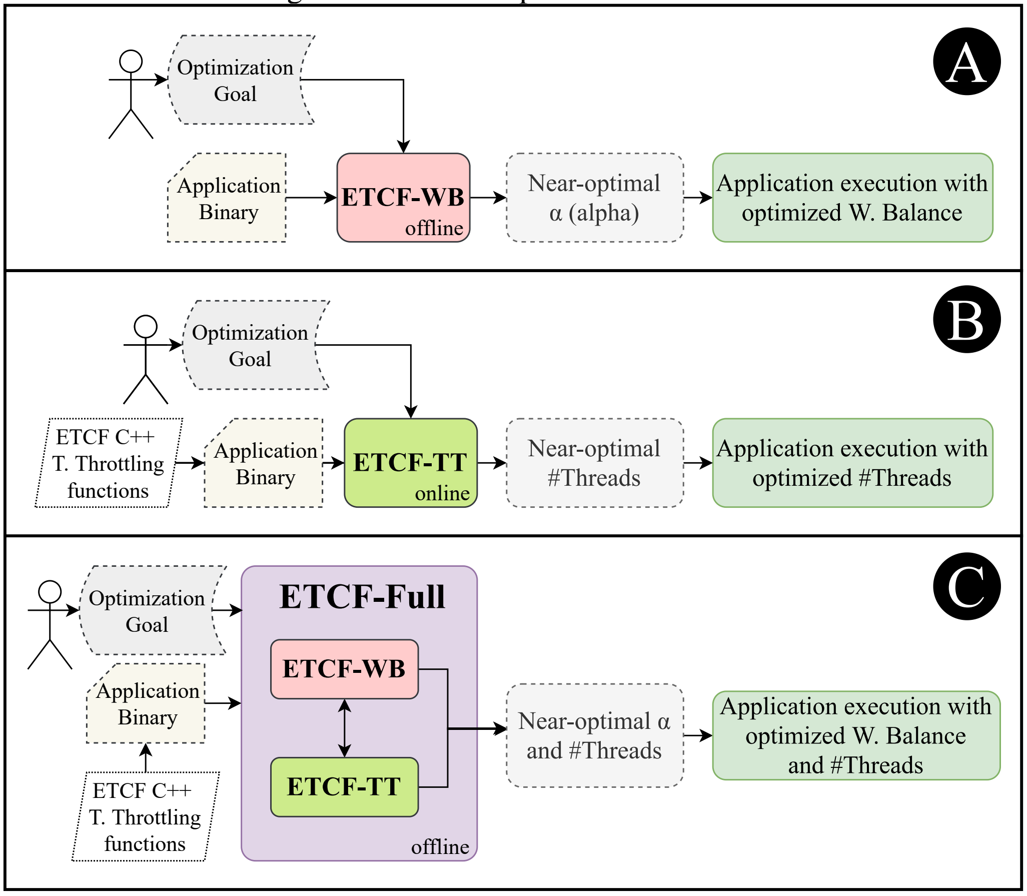
Source: the author

4.2.1.1 ETCF Workload Balance

ETCF-WB optimization mode optimizes the workload balance by using the data partitioning collaborative technique (HUANG et al., 2019). As shown in Figure 4.6-A, this mode requires the application binary and the optimization goal as inputs. The applications' kernels need to be implemented in OpenCL (STONE; GOHARA; SHI, 2010), which allows the application to partition the data to the different heterogeneous devices. To dynamically adjust the workload balance at runtime, it will be required modifications on the OpenCL API. In order to avoid such modifications, ETCF-WB comprises a software tool that runs the application passing different values of α as argument, which defines the amount of data assigned across the device (CPU/FPGA), while keeping a fixed #Threads with the maximum supported by the CPU. In this way, our framework can perform its search only by varying the application's α argument at each search iteration (application execution), without binary modifications. At each iteration energy and delay measurements are collected to evaluate the metrics, according to the optimization goal (i.e. execution time when using performance as optimization goal).

Given that some applications may demand huge completion time, we use an approach to quickly converge into a satisfactory solution, avoiding a combinatorial explosion of testing any value ranging from 0 to 1. Figure 4.7 illustrates how our method is able to rapidly find a near-optimal workload balance. In the first and second iteration, our search algorithm runs the application using a workload balance of 0.1 and 0.9 (α),

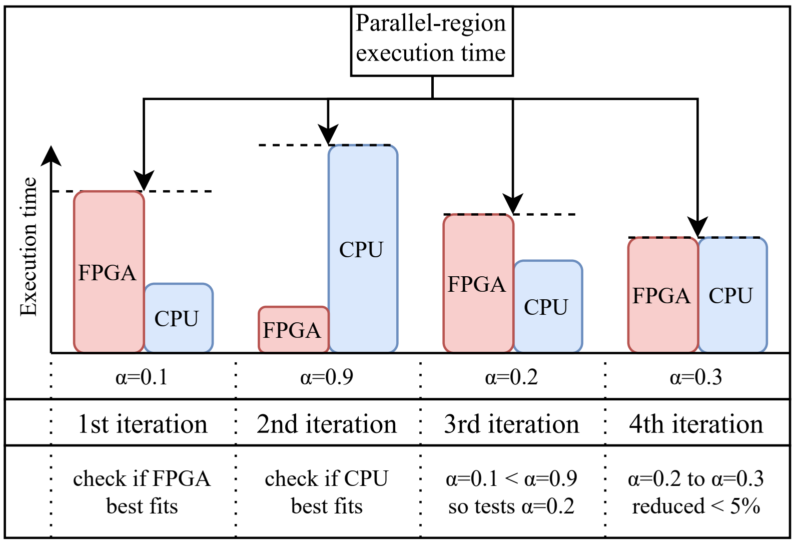
Figure 4.6: ETCF Optimization Modes.


A
Optimization Goal
Application Binary
ETCF-WB
offline
Near-optimal α (alpha)
Application execution with optimized W. Balance
B
Optimization Goal
ETCF C++ T. Throttling functions
Application Binary
ETCF-TT
online
Near-optimal #Threads
Application execution with optimized #Threads
C
Optimization Goal
Application Binary
ETCF C++ T. Throttling functions
ETCF-Full
ETCF-WB
ETCF-TT
offline
Near-optimal α and #Threads
Application execution with optimized W. Balance and #Threads

Source: the author

respectively, meaning that 10% of the data is assigned to the CPU and 90% to the FPGA, and vice versa. These initials steps aim at detecting if one of the devices best fits the application at hand. Then, we compare the results of both α values to see if they reached the stop condition. The stop condition is achieved when a near-optimal workload balance is found. For practical reasons, we consider a 5% difference on the optimization goal (e.g., performance, energy, etc) as the stop condition. For example, in a performance-targeted scenario, if a measurement at a given α presents only 5% difference in execution time compared to the previously tested α, the algorithm ends the search. However, in case neither initial values (i.e., 0.1 or 0.9) reached near-optimal workload balance, the next alpha to be tested is defined by the alpha that provided the better metric. If 0.1 has a better metric than 0.9, the algorithm will test $\alpha = 0.2$; else it tests $\alpha = 0.8$. The values are progressively decreased/increased by 0.1 until they reach the stop condition.

Figure 4.7: ETCF Workload balance.



Source: the author

4.2.1.2 ETCF Thread Throttling

Because CPU and FPGA can largely vary from one system to another, selecting the number of CPU threads at programming time is not trivial, as shown in Section 3.3. Given that, ETCF-TT optimization mode uses ETCG's run-time ETCG approach proposed in Section 4.1, which can automatically selects the number of CPU threads considering performance and energy the system presents at execution time, which may not be known by the programmer.

4.2.1.3 Full ETCF

Since the optimal workload balance is influenced by the performance that the CPU offers, our workload balance search provides its best performance when used together with ETCF-TT. In light of this, ETCF also offers a full mode where both ETCF-WB and ETCF-TT in a synergistic way. Full ETCF mode comprises the offline Workload Balance mode using an application already optimized using ETCF-TT approach. Thus, it requires as input the optimization goal, as the previous modes, but also a C++/OpenCL application implemented inserting our ETCF-TT C++ functions. In this way, Full ETCF can acquire the best performance each workload balance can offer at its ideal #Threads.

To enable the Full ETCF mode, the administrator just needs to use the ETCF-TT set of functions on his application source-code, then insert the compiled program into the ETCF-WB tool, which runs the application setting different workload balances (alpha). This way, both *Thread Throttling* and Workload Balance approaches are be applied together, once ETCF-WB runs each iteration of the workload balance search using the near-optimal #Threads for each given alpha.

4.2.2 Methodology

To evaluate ETCF, we used the same benchmarks from Chapter 3, and the CPU and the FPGA specified in Table 3.1, running upon the same OS and drivers, as well as using the same compiler and tools to collect power data.

4.2.2.1 Evaluation Scenarios

To validate our proposal, we used the following scenarios:

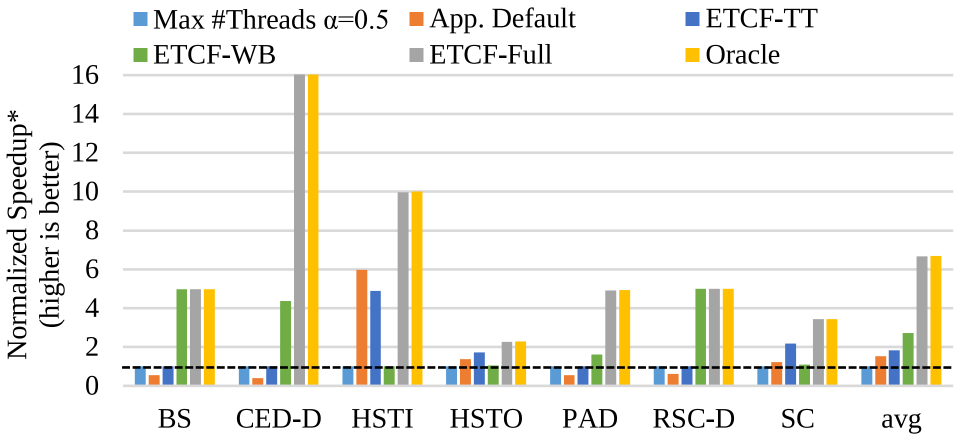
- Max #Threads, used as our baseline, where #Threads is set to the maximum number of supported threads that can be executed concurrently according to the available hardware in (in our case, 64 cores with SMT can run 128 threads in parallel), which is the default approach in many other parallel APIs, such as OpenMP. We considered a hardware-unaware approach for the workload balance, so we kept an evenly balance of the workload among the devices ($\alpha = 0.5$);
- App. Default, where the default #Threads and α are kept (i.e. defined in the app. programming code);
- Oracle, which represents the optimal scenario where #Threads and α are gathered offline from an exhaustive search;
- ETCF-TT: ETCF *Thread Throttling* Mode, where the #Threads is defined after applying our Hill-Climbing based algorithm as described in Section III, and the workload balance is kept in $\alpha = 0.5$;
- ETCF-WB: ETCF Workload Balance Mode, where the #Threads is set to the maximum number of supported threads, and the workload balance is produced by the ETCF offline phase;
- ETCF-Full, ETCF-TT + ETCF-WB, where the #Threads is defined after applying our Hill-Climbing based algorithm as described in Section III, and the workload balance is produced by the ETCF offline phase.

The scenarios described above were evaluated considering three different metrics: performance, energy consumption, and EDP. Additionally, we examined the overhead imposed by the proposed search algorithm by comparing the time to solution (i.e., time to define the optimal number of threads and workload balance) of ETCF framework with the exhaustive search. We further investigated the influence of the #Threads changing the workload balance, showing the impact on the EDP presented by each #Threads under these circumstances.

4.2.3 Results

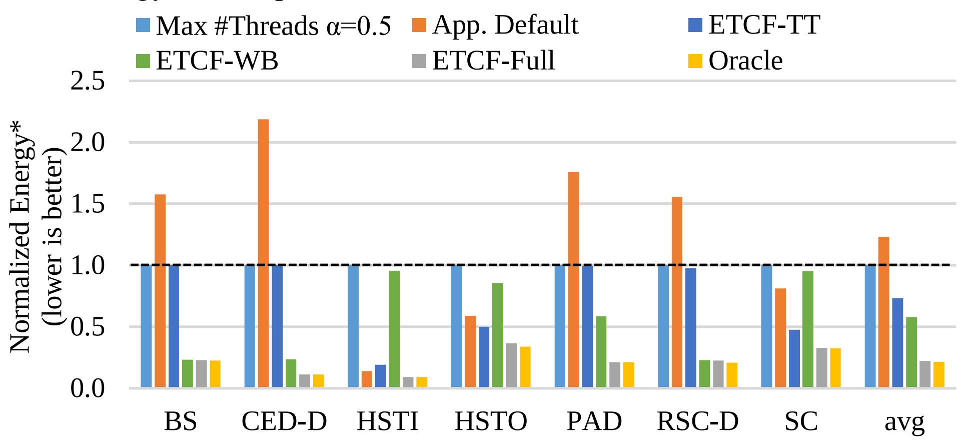
In this subsection, we configured ETCF with all available optimization goals to evaluate ETCF-TT, ETCF-WB, and ETCF-Full modes. Figures 4.8, 4.9, and 4.10 show the experimental results when ETCF framework has distinct optimization goals: performance (A); energy consumption (B); and EDP (C). We evaluated each metric considering the scenarios described in Section 4.2.2.1, where gains are normalized w.r.t *Max #Threads* baseline scenario, which uses α=0.5 (50% of workload balance). For the sake of readability, we used logarithmic scale in the Figure 4.10. Appendix B presents tables with the experiments collected for the ETCF evaluation.

Figure 4.8: Performance improvements w.r.t the baseline *Max #Threads* (α=0.5).



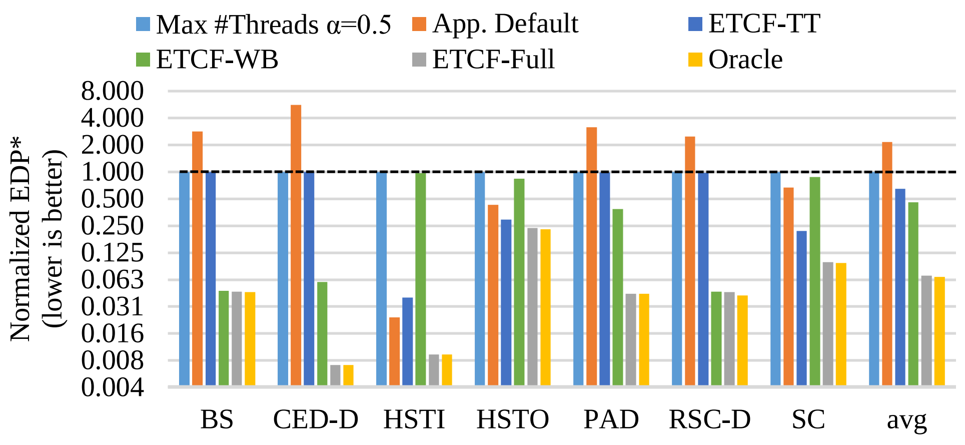
Source: the author

Figure 4.9: Energy consumption normalized w.r.t the baseline *Max #Threads* (α=0.5).



Source: the author

Figure 4.10: EDP normalized w.r.t the baseline *Max #Threads* (α=0.5).



Source: the author

4.2.3.1 ETCF-TT Mode

Compared to *Max #Threads* baseline, our *Thread Throttling* approach *ETCF-TT* is able to provide an average speedup of 1.83x, with 27% and 35% reduction in energy consumption and EDP, respectively. As mentioned in Section 4.1.3, the significant overhead of the thread creation process causes performance penalties when the number of threads increases. This makes the CPU taking longer to run its portion of kernels' data, while the FPGA remains idle waiting for the CPU data to be processed. Since *Max #Threads* takes longer to execute than scenarios with a lower number of active threads, it spends more energy, since all CPU threads remain active dissipating power.

The *App. Default* scenario can provide performance and energy benefits in some applications such as HSTI, HSTO, and SC. This behavior is due to low thread scalability

(LORENZON et al., 2018) in these applications. On the other hand, other applications such as CED-D and PAD present a lower performance using the default #Threads. This occurs because the static #Threads defined by the programmer may not perform well in all systems. In this scenario, a dynamic approach such as *ETCF-TT* is able to fit the application for the hardware at hand, providing performance and energy improvements on most of the applications.

4.2.3.2 ETCF-WB Mode

In terms of performance (Figure 4.8), on average, our workload balance approach *ETCF-WB* outperforms the baseline *Max #Threads* in 2.73x, while consuming 42% less energy and with a 54% reduction in EDP (Figures 4.9 and 4.10). As shown in Figure 4.10, scenarios with statically defined workload balance - such as *Max #Threads*, *App. Default*, and *ETCF-TT* - do not provide the lowest EDP values for most benchmarks, since the power of each device of the heterogeneous system (CPU/FPGA) can drastically vary with the size of workload assigned. In this way, applications may execute faster in one device than the other, which leads the faster device to remain idle until the other finishes its processing. Similarly, if one device is more energy-efficient than the other for some applications, it is worth assigning more data to the most efficient device to balance the workload.

It can be observed that most of the experimented applications had an ideal workload balance different from the baseline (α=0.5), which led *ETCF-WB* to have similar performance and energy results to the *Oracle* in applications such as BS and RSC-D.

4.2.3.3 ETCF-Full Mode

Given the improvements previously shown by ETCF-TT and WB modes, our framework can further improve performance or energy consumption for applications where both TT and WB approaches bring gains.

Figures 4.8, 4.9, and 4.10 show the gains provided when our *ETCF-Full* approach is applied. It presents an average 6.67x performance improvement compared to the baseline *Max #Threads*, while consuming 78% less energy and 93% lower EDP. These benefits come from a well-balanced workload and properly #Threads set.

To investigate where such gains came from, we compare *ETCF-Full* to *ETCF-TT* and *ETCF-WB*. Some applications, like HSTO, have similar improvements for *ETCF-*

Full and *ETCF-TT*, which induces that *Thread Throttling* is responsible for the majority of the improvements. Other applications, such as BS and RSC-D, have similar improvements on *ETCF-WB* and *ETCF-Full*, showing that most of the improvements come from Workload Balance. However, the other four applications have greater improvements using *ETCF-Full* than a decoupled use of *ETCF-TT* and *ETCF-WB*, showing the benefits of the synergistic optimization of both approaches.

4.2.3.4 ETCF-Full x Oracle Evaluation

As shown by the Figures 4.8, 4.9, and 4.10, compared to the *Oracle*, *ETCF-Full* presents only 0.9%, 0.8%, and 0.2% performance, energy, and EDP degradation, showing the effectiveness of ETCF on detecting near-optimal workload balance and number of threads. Although providing slightly better results compared to *ETCF-Full*, the *Oracle* demands huge time to solution (i.e., time to define the optimal number of threads and workload balance), as shown in Table 4.3. Taking on average only 3.32% of the time compared to the *Oracle*, *ETCF-Full* achieves solution close to the optimal (given by the *Oracle*). When comparing the *ETCF-WB* to the *Oracle*, it only takes 1% or less of the *Oracle*'s Time to Solution fraction to produce a solution, considering all applications. *ETCF-WB*, even though being an offline method, can produce solutions requiring few applications executions. Meanwhile, the purely online *ETCF-TT* can provide the near-optimal #Threads taking only 0.057% of *Oracle*'s search time. The *ETCF-TT* demands, on average, 6 kernel iterations to converge to a near-optimal #Threads, while the *Oracle* requires 1152 application executions to define the solution.

Table 4.3: Time to solution (seconds).

App.	Total Execution Time	Oracle Search Time	ETCF-TT Serach Time	ETCF-WB Search Time	ETCF-Full Search Time
BS	0.549	2048.242	0.215	15.824	4.001
CED-D	5.593	17088.238	0.060	134.977	29.718
HSTI	0.230	408.522	0.017	4.193	1.071
HSTO	0.509	829.780	0.069	8.073	2.393
PAD	0.358	923.354	0.073	7.459	2.048
RSC-D	1.971	9565.313	1.147	74.749	17.327
SC	0.322	513.563	0.025	4.782	1.327
Average	1.362	4482.430	0.229	35.722	8.269

Source: the author

4.3 Discussion

In this chapter we proposed ETCG, an approach to automatically determining a near-optimal number of threads for the CPU in CPU-GPU collaborative environments. In contrast to the optimal exhaustive search, ETCG achieves similar solutions in a feasible time. Compared to the static use of the #Threads, it reduces EDP by up to 73%.

Moreover, we proposed ETCF, a configurable framework to improve performance, energy, and EDP in CPU-FPGA collaborative environments by determining a near-optimal workload balance between CPU and FPGA and the number of threads for the CPU. As opposed to the optimal exhaustive search, ETCF achieves similar solutions in a feasible time. Compared to the static use of the maximum number of threads, it increases performance by 6.67x, while reducing energy and EDP by up to 78% and 93%.

5 ON THE BENEFITS OF APPLYING CPU *THREAD THROTTLING* AND *HLS-VERSIONING* IN CPU-FPGA TASK-COLLABORATIVE ENVIRONMENTS

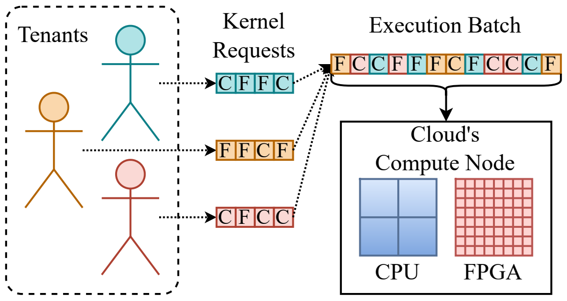
This chapter investigates the impact of applying optimizations on both CPU and the FPGA devices in CPU-FPGA Task-Collaborative Environments. We use a multi-tenant Cloud service as our object of study, in which we perform a DSE to evaluate the benefits of applying *Thread Throttling* on the CPU kernels and *HLS-Versioning* on the FPGA kernels.

In the preceding chapters, the optimizations were applied to single applications that use the data-collaborative execution model, which enables an accessible workload balance by proportionally partitioning the data among the devices (setting the alpha parameter). In contrast, the present chapter exploits the task-collaborative cloud environment where distinct CPU-only and FPGA-only kernels run concurrently - task and data-collaborative execution models already discussed in Chapter 2. In such a scenario, it is not trivial to workload balance kernels, once they present very different characteristics w.r.t how they perform over the heterogeneous hardware, requiring resource provisioning algorithms to assign the kernels to the devices. Such strategies are out of the scope of this dissertation, in which we stick to enhancing the performance and energy of the collaborative execution by optimizing the individual kernels' execution, considering pre-established batches of kernels to be executed.

In multi-tenant Cloud Environments, the infrastructure resources, such as CPUs and FPGAs, are shared between clients (tenants) using the Acceleration-as-a-Service (AaaS) model (CHEN et al., 2014), as described in Figure 5.1. In such environments, the tenants' kernels compete over the CPU and FPGA resources while presenting execution precedence restrictions among them, which impedes them to run while another is being served with the CPU or the FPGA resources, so they are mutually affected by their execution time. Applying optimizations, such as *Thread Throttling* and *HLS-Versioning*, vary the time and energy taken to execute the CPU and FPGA kernels, offering the opportunity to enhance these metrics considering the overall execution of sequences of kernels. This way, the synergistic optimization of the CPU and FPGA kernels may help to achieve the tight performance and energy requirements of cloud environments.

The remainder of this Chapter is organized as follows. First, Section 5.1 introduces the methodology used in the evaluation, specifying the execution environment and set of benchmarks. Then, Sections 5.2 presents the experimental evaluation investigating

Figure 5.1: Cloud's Acceleration-as-a-Service model.



Source: the author

the optimization opportunities regarding the *Thread Throttling* and *HLS-Versioning* techniques applied to our set of benchmarks. Section 5.3 presents a experimental grounding considering the multi-tenant cloud environment. Finally, section 5.4 shows the results acquired from the experiments.

5.1 Methodology

5.1.1 Execution Environment

The execution environment used to evaluate this approach was the same used to evaluate ETCG and ETCF, already described in Section 3.1.1. That way, we used the CPU and the FPGA specified in Table 3.1, running upon the same operating system (OS) and drivers, as well as using the same compiler and tools to collect power data. For FPGA kernels, specifically, we considered the Clock Frequency achieved in (CONG et al., 2018). We used bash scripting to run all the experiments shown in the present chapter.

5.1.2 Benchmarks

We used benchmarks from the Rodinia suite (CHE et al., 2009), since it comprises applications that exploit CPU thread parallelism using OpenMP (DAGUM; MENON, 1998), and exploit FPGA parallelism via high-level synthesis optimizations with kernels implemented using the OpenCL standard (STONE; GOHARA; SHI, 2010). Contrarily to the Chai suite, used in the experiments from the prior chapters, the Rodinia suite offers well-known open-source HLS versions for their kernels. In the experiments, we used the kernels presented in the table 5.1.

Table 5.1: Rodinia benchmarks.

Benchmark	Description
Backprop	Back Propagation is a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network.
CFD	The CFD solver is an unstructured grid finite volume solver for the three-dimensional Euler equations for compressible flow.
Kmeans	K-means is a clustering algorithm used extensively in data-mining and elsewhere, important primarily for its simplicity.
KNN	K Nearest Neighbor finds the k-nearest neighbors from an unstructured data set.
LavaMD	LavaMD calculates particle potential and relocation due to mutual forces between particles within a large 3D space.
NW	Needleman-Wunsch is a non-linear global optimization method for DNA sequence alignments.
PathFinder	PathFinder uses dynamic programming to find a paths on a 2-D grid from the bottom to the top row with the smallest accumulated weights.
SRAD	SRAD is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations.

Source: the author

For *Thread Throttling* configuration, OpenMP provides an easy control of how many concurrent threads will be launched to the execution via an OS environmental variable, allowing us to explore the performance and energy consumption each combination of #Threads has to offer. The CPU used in our experiments supports up to 128 concurrent threads, so this is the amount of versions that will be exploited in CPU kernels. Considering the *HLS-Versioning*, FPGA Xilinx Vivado Tool also provides accessible parallelism exploitation using High-Level Synthesis optimization pragmas, enabling different techniques that result in kernel designs with variant trade-offs between performance and energy. In our experiments, we use the 6 versions available for each kernel from Rodinia-HLS repository (CONG et al., 2018), as depicted in Table 5.2.

Table 5.2: Rodinia FPGA HLS versions.

Version	Description
V1 – Baseline	It comprises the ported kernels from the CPU implementation without applying any optimization technique.
V2 – Tiling	It tiles the high-level code, transfer a data tile from main memory to on-chip BRAM, and cache a data tile on-chip for later accesses.
V3 – Pipeline	It implements the iterations of loop statements of the code into a hardware pipeline.
V4 – Unroll	It unrolls loops without data dependencies between iterations, implementing each one in parallel FPGA processing elements.
V5 – Double Buffer	By duplicating on-chip BRAM data, this optimization avoids read-write dependencies, enabling the overlapping of computation and memory accesses of different tiles.
V6 – Coalescing	It increases the bit width of memory access (coalescing) and increases access length (bursting) by bonding several narrow data into a single wider data.

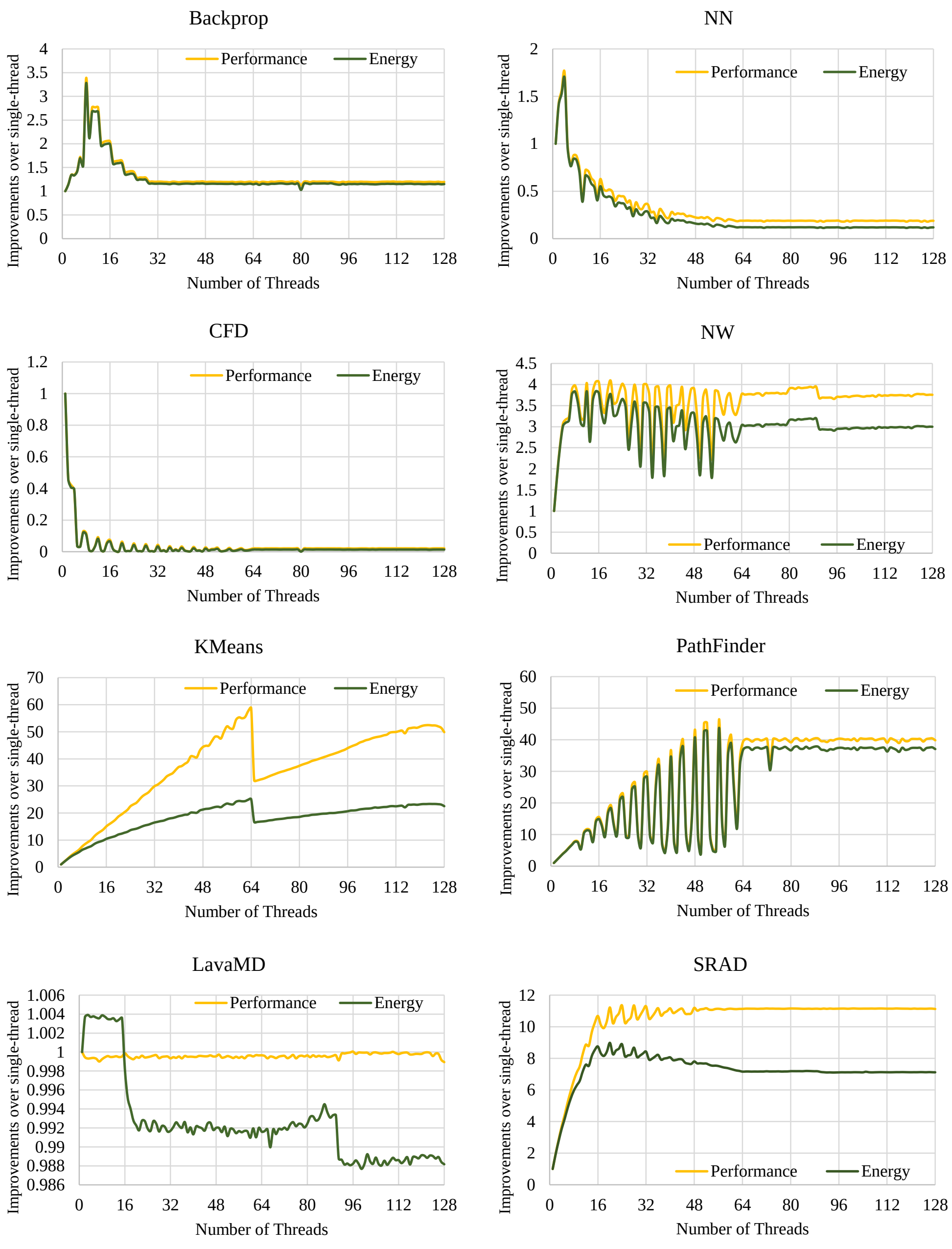
Source: the author

5.2 *Thread Throttling* and *HLS-Versioning* opportunities

Before heading into the more complex experiments considering the multi-tenant cloud execution, in this section we characterized the optimization opportunities of the set of kernels we used regarding the *Thread Throttling* and *HLS-Versioning* techniques. As it occurs in Cloud environments, we collected kernel's run-time execution time and energy consumption by running each one standalone with all possible configurations (i.e. running CPU kernels with all #Threads supported by the hardware and FPGA kernels with all HLS optimizations available). Is important to point out that the evaluation performed in this section represents a non-collaborative execution, since we measured the optimization opportunities by running the kernels in CPU and FPGA in isolation, without any interference between them.

Figure 5.2 shows the performance and energy improvements of all kernels w.r.t. the single-thread execution in the Y-axis, and the number of threads used to run the kernels in the X-axis. At a first glance, we can notice that the kernels present divergent behaviors when increasing the #Threads. The Kmeans kernel present a great scalability, achieving almost 60x performance improvements when using 64 threads, with a slightly lower performance when SMT is enabled between 65 to 128 threads. Similarly, PathFinder showed more than 40x performance and energy gains using 56 threads, although the workload of this kernel achieves greater performance when using an even #Threads, a behavior already investigated by LORENZON et al.. The NW and SRAD kernels also presented performance and energy improvements with the TLP exploitation, despite stagnating the gains when using more than 16 threads. Backprop and NN reach best performance and energy with only 8 threads. The LavaMD kernel has almost no benefit from TLP exploitation. Finally, CFD presented optimal performance and energy with single-thread execution, showing degradation when using more than one thread. With such evaluation, we can observe that *Thread Throttling* is capable of providing performance and energy improvements for all the kernels, since none of them have optimal efficiency when using the default maximum #Threads available.

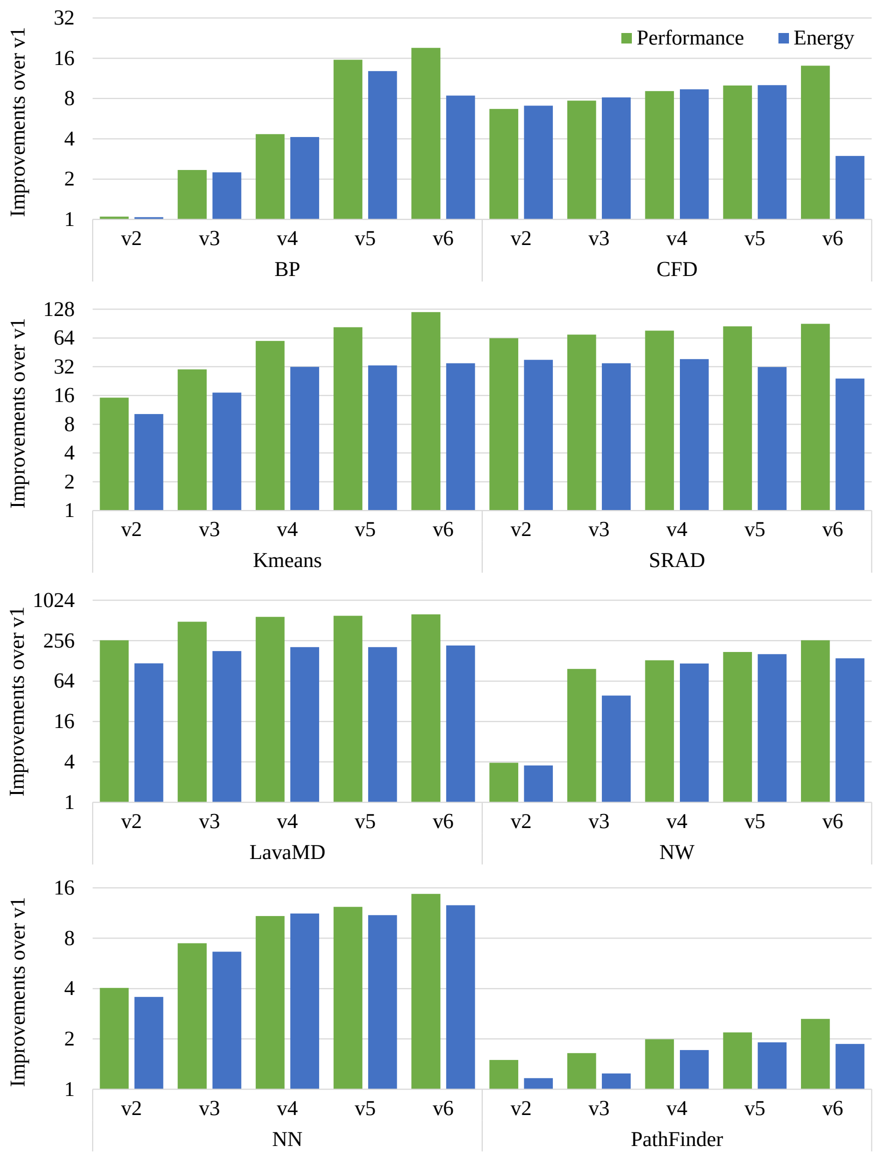
Figure 5.2: TLP scalability of Rodinia CPU kernels.



Source: the author

Regarding the FPGA execution, we evaluated the potential improvements of applying *HLS-Versioning* to the kernels. Figure 5.3 shows performance and energy gains of HLS-generated versions of the kernels from our set of benchmarks using different HLS optimization techniques. In the chart, the X-axis indicates different HLS versions for each kernel, while the y-axis denotes the improvements with regard to performance and energy compared to the non-optimized HLS version (v1). The experiments consider all the 6 HLS-versions available for each of the Rodinia kernels (CONG et al., 2018), which present incremental optimizations to each version. We can point out that not only different kernels benefit from different optimizations, but also that the best choice for performance is not always the best one for energy for the BP, CFD, SRAD, and NW kernels.

Figure 5.3: *HLS-Versioning* applied to Rodinia FPGA kernels.



Source: the author

One can notice that some kernels have more benefits when applying the CPU *Thread Throttling* optimization, such as CFD, PathFinder, and NN, while other kernels such as Kmeans and LavaMD present more benefits from the use of *HLS-Versioning* on the FPGA, showing the complementarity of the optimizations considering the global scenario.

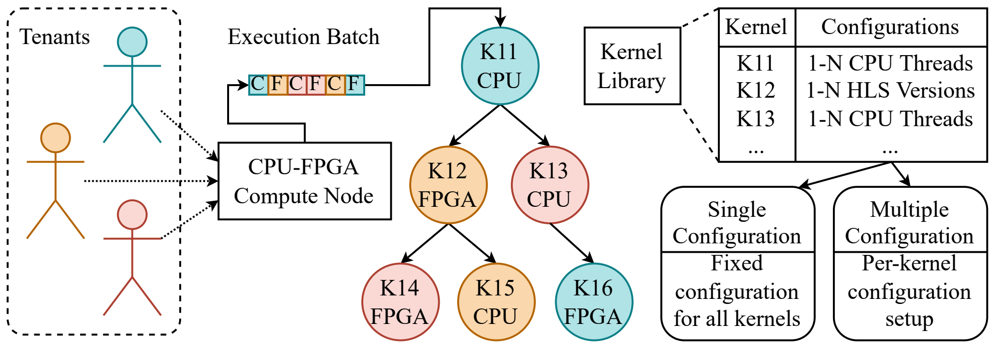
In this way, exploring the benefits brought by both optimizations may bring significant gains in CPU-FPGA systems. However, the problem of finding the best combination of where to execute (CPU or FPGA) and which version to use (with the possible configurations of *Thread Throttling* or HLS) is further aggravated considering that sequences of many kernels must be executed. In multi-tenant Cloud services, used as our object of study, several tenants that share the infrastructure resources make multiple kernel requests, which have different priorities - further increasing the number of possible solutions.

5.3 Experimental Grounding

The AaaS model (CHEN et al., 2014) used in multi-tenant Cloud Environments determines the execution of their tenants' kernels according to priorities and precedence restrictions, so the batches from Figure 5.1 are not necessarily executed in a straightforward First-in, First-out (FIFO) fashion. Instead, such restrictions lead to workloads in the format of Direct Acyclic Graphs (DAGs), in which each node of the graph represents an individual kernel (HILMAN; RODRIGUEZ; BUYYA, 2020), as Figure 5.4 illustrates. Distinct DAGs present independent kernels that can execute concurrently. AaaS Cloud's Kernel Library keeps different configurations for each kernel, along with their execution time and power dissipation (CHEN et al., 2014; FAHMY; VIPIN; SHREEJITH, 2015). These metrics are gathered beforehand by the Cloud's managers, permitting them to access this data to anticipate the kernels' run-time behavior and select the most appropriate configuration when launching kernel. For instance, our OpenMP CPU kernels can be configured to execute with different #Threads via the *OMP_NUM_THREADS* environment variable, and our set of FPGA kernels can be implemented using the different bitstreams generated from the *HLS-Versioning*, which result in different execution times and energy consumption, as previously shown in Section 5.2.

Given that, in this chapter we present a DSE over the CPU and FPGA configurations (i.e., not affecting the workload DAG), once our study is orthogonal to scheduling

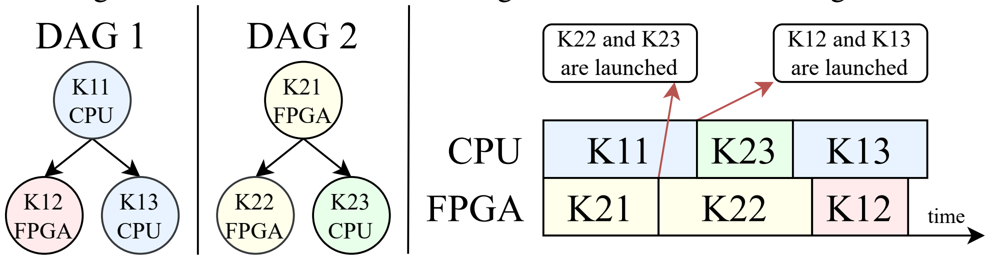
Figure 5.4: Cloud's DAG workload scheme and Kernel Configuration.



Source: the author

works, so pre-established DAGs are considered. For this reason, we model the execution using binary tree-like DAGs, while walking through the nodes with a conventional Breadth-First Transversal (BFS) algorithm. Our BFS algorithm starts at the root node and launch all nodes at a given level to the CPU and FPGA execution queues, using a left to right order. When a certain node concludes its execution, its children are launched using the same procedure. Figure 5.5 exemplifies how kernels are assigned to the collaborative devices. Two DAGs are presented in the example, which means their execution are totally independent, and their kernels dispute over the shared CPU and FPGA resources. Because of that, the launch of K11 and K21 occurs concurrently, while K22 and K23, which depends on K21, are launched just after K21 finish its execution.

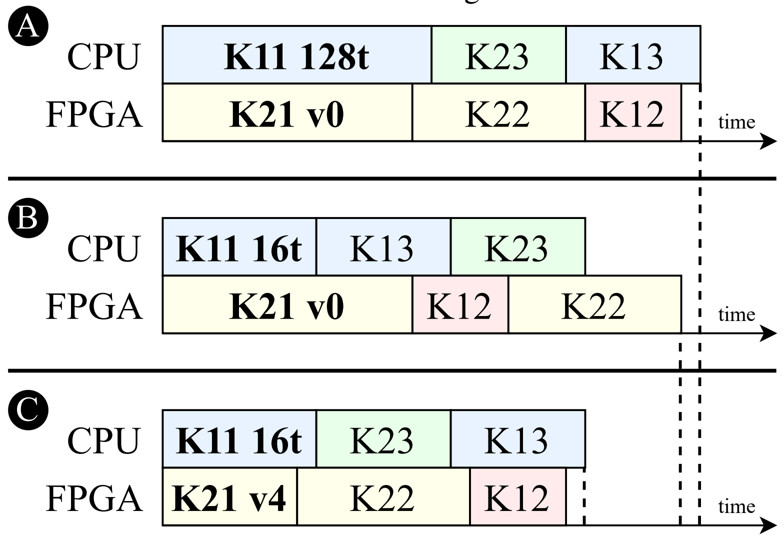
Figure 5.5: Kernel allocation using our conventional BFS algorithm.



Source: the author

Figure 5.6 illustrates how an appropriate configuration of #Threads and HLS version can benefit the entire DAG execution, using the same kernels from Figure 5.5 as an example. In Fig 5.6-A, the default #Threads and HLS version are used. In Figure 5.6-B *Thread Throttling* is applied to the K11 version, resulting in a reduction of the DAG execution time, now limited by the FPGA execution time. Meanwhile, Figure 5.6-C shows *Thread Throttling* applied to the kernel K11 but also *HLS-Versioning* employed in the kernel K21, leading to a further reduction in execution time.

Figure 5.6: How different CPU/FPGA configurations affect overall execution.



Source: the author

To fully explore (i.e. find the right configuration for all kernels) the design space offered by our experiments, an exhaustive search would be necessary. However, exploring all possible combinations of kernel's versions would result in a combinatorial explosion when expanding the DAGs' depth, as presented in Table 5.3, which considers our set of kernel versions and our hardware setup. Given that, the exponential time complexity of exhaustively searching for the solutions rapidly becomes unfeasible when increasing the number of nodes on the DAG.

Table 5.3: Search complexity.

DAG depth	Nº of nodes	Nº of combinations	Time to solution
2	6	2.1e4	14 minutes
3	10	1.6e7	37 days
4	14	1.3e10	329 years
5	18	9.6e12	974 millennia

Source: the author

In order to avoid this issue, we have limited our search space by exploiting the kernel configurations, considering two approaches to select the kernel's configurations: Single Configuration and Multiple Configurations.

The Single Configuration approach consists in a DSE over the combinations of CPU and FPGA configurations, using a fixed #Threads and HLS version for all nodes in the DAG. For instance, we collect execution time and energy consumption of the system when executing the DAG with all CPU nodes using 1 thread, and all FPGA nodes using HLS version 1. Then we repeat the process until testing every combination of #Threads and HLS versions. This way, our search space is proportional to the limited kernel versions of the Cloud library, resulting in a linear search time complexity. The Single Configuration DSE is used in this Chapter as a comparison to the Multiple Configuration solution.

Meanwhile, in the Multiple Configurations approach we select the kernels' configurations individually, using a local search algorithm that aims at finding the kernel's version that provides the lowest EDP. For that, the algorithm orders the versions by the EDP offered by each one, then selects the one that provides the lowest EDP, balancing the trade-off between performance and energy consumption. It is also important to point out that, despite using local EDP measurements when selecting the configuration, Experimental Results in Section 5.4 will present Performance, Energy, and EDP on the whole DAG execution, showing how the proper configuration of the kernels impacts the global execution.

5.3.1 Evaluation Setup

We implemented our set of experiments inside a Linux bash script, which launches the kernels for execution considering a pre-established order defined by the DAGs. The DAGs were structured using the previous descriptions of the present section, in which we randomly assigned the kernels from our set of benchmarks into the nodes of 100 parallel DAGs, for a practical reason, with up to 100 depth levels each, where the number of depth levels is also randomly defined by our script. To exploit the Single Configuration approaches, we ran the given set of DAGs using the different combinations of #Threads and HLS-versions. While for Multiple Configurations approach we search for the version for the best EDP for the kernel of the given node, then launches this optimal version for execution. For both approaches we used all the available CPU and FPGA versions for our set of kernels, where each CPU kernel offers 128 possible configurations of #Threads (from 1 to the maximum #Threads supported), while FPGA nodes offer 6 versions utilizing different HLS optimizations. To build a large sample of experiments, we repeated this process 10.000 times, also for a practical reason, each one using distinct DAG structures with different kernel assortments over the DAGs, and used the average execution time and energy consumption in our evaluations.

In order to facilitate the presentation of our experimental results, we divided the results from the *Single* and *Multiple Configuration* approaches into different experimental scenarios. Considering the *Single Configuration Scenarios*, #Threads and HLS version are kept fixed for all CPU and FPGA nodes. While the *Multiple Configuration Scenario* aims at investigating the influence of selecting the best *Thread Throttling* and *HLS-Versioning* configurations for each node of the graph (i.e. each kernel may have a different configuration w.r.t. the #Threads and HLS version). Our Single Configuration Scenarios are given as follows:

- Baseline - the default in CPU-FPGA environments, in which OpenMP uses the maximum #Threads available for the CPU - 128 threads in our setup -, and FPGA HLS is performed without any optimizations such as *Array Partitioning*, *Loop Unrolling* or *Loop Pipelining* - version 1 (v1) in our experiments;
- Single Configuration - comprises the execution of all DAGs (CPU and FPGA kernels) using fixed number of Threads and a unique HLS version.

Our Multiple Configuration Scenarios are given as follows:

- Multiple Configurations CPU-TT Only - only *Thread Throttling* is applied aiming at minimizing the EDP resulted from the CPU nodes, while the FPGA HLS is still performed without any optimization;
- Multiple Configurations HLS-V Only - only *HLS-Versioning* is performed at the FPGA kernels and the version which produces the best EDP results is selected for the execution of the FPGA nodes. The *Thread Throttling* configuration is kept in 128 threads;
- Multiple Configurations CPU-TT + HLS-V - both CPU *Thread Throttling* and *HLS-Versioning* are performed on all DAG nodes to minimize EDP in the CPU-FPGA Collaborative environment.

5.4 Results

This Section presents the results obtained through the DSE performed in our experiments. Subsection 5.4.1 shows the improvements produced using Single Configurations of #Threads and HLS versions across all nodes. After, Subsection 5.4.2 presents the gains yielded by Multiple Configurations of #Threads and HLS versions for each individual node. Finally, Subsection 5.4.3 compares Single and Multiple Configuration approaches, discussing the strengths and weaknesses of each. Additionally, appendix C presents tables with the experiments collected for this chapter.

5.4.1 Single Configuration Scenarios

Table 5.4 presents the EDP improvements of each combination of #Threads and HLS version over the Baseline (values under 1 mean EDP degradation). Column 1 (HLS Version 1) presents all configurations with the non-optimized HLS version. As can be noticed, the configuration with 16 threads provided the best average solution across the nodes, with 25.8% of EDP gains. Instead, the last row of the table show us the benefits provided by the HLS optimizations, since #Threads are kept at the maximum hardware threads available (i.e., 128 threads). For 128 threads, HLS version 4 provided the best trade-off between performance and energy, with 3.276x EDP gains. Considering all results, the sweet spot of this scenario was using 16 threads and HLS version 4 in the CPU and FPGA nodes, respectively, producing a 9.348x EDP gain over the baseline. Such behavior occurs because idle times can occur in one device when other dependent kernel is running in the other device, so using kernel versions with highest EDP gains individually does not necessarily reflect on the highest overall EDP improvements. Moreover, despite providing, on average, the best single configuration result, some kernels may have different optimal #Threads and HLS version than the average, giving room for improvements that will be show next using Multiple Configurations in the following section.

Table 5.4: EDP gains ($\times$) using Single Configuration exploitation.

	HLS Version					
#Threads	1	2	3	4	5	6
1	0.273	0.274	0.274	0.274	0.274	0.274
2	0.860	0.869	0.870	0.870	0.870	0.870
4	1.092	2.497	2.499	2.501	2.501	2.500
8	1.255	7.632	7.644	7.656	7.654	7.648
16	1.258	9.318	9.334	9.348	9.346	9.338
32	1.162	5.718	5.726	5.733	5.732	5.728
64	1.009	3.351	3.354	3.357	3.357	3.355
128	1.000	3.270	3.273	3.276	3.275	3.274

Source: the author

5.4.2 Multiple Configurations Scenarios

Figure 5.7 shows the benefits from using multiple configurations, in which *Thread Throttling* and *HLS-Versioning* are applied to the individual kernels present on the DAGs. The y-axis shows the improvements over the baseline (in terms of Performance, Energy, and EDP), while the x-axis displays each multiple configuration scenario. Standard deviations inherent to the samples are presented alongside the whiskers at the top of the bars. For the sake of readability the chart is presented in logarithmic scale.

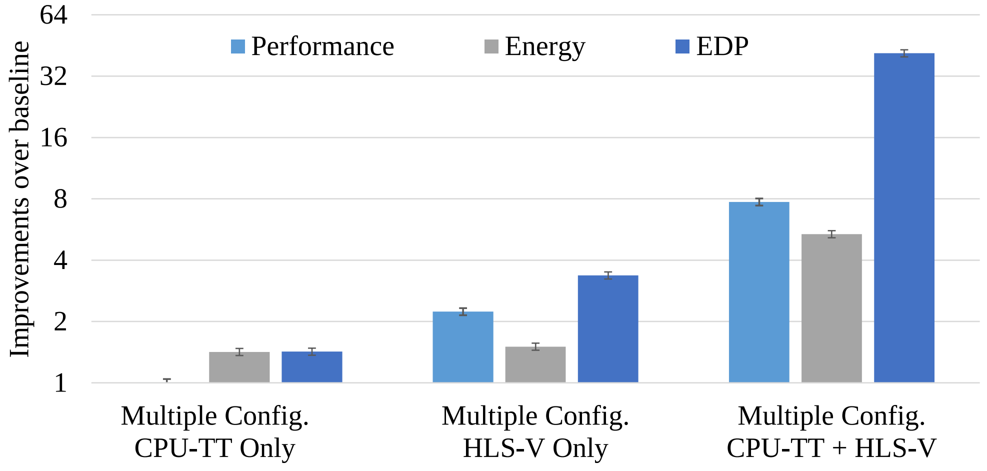
When considering the CPU-TT Only scenario, execution time of the DAGs are not improved compared to the baseline, since applying *Thread Throttling* did not speed up these applications. In contrast, Energy consumption is usually reduced when *Thread Throttling* is applied, since it greatly decreases the power dissipation at only small penalty in execution time. In summary, *Thread Throttling* provided 1.418x EDP improvements on the DAGs' overall execution, surpassing the 1.258x gain provided by using a single configuration for the #Threads on all CPU nodes, as shown in Section 5.4.1.

Unlike the CPU-TT Only, the HLS-V Only scenario boosted the performance in 2.236x over the baseline. This behavior arises from reductions in execution time for kernels that can benefit from HLS optimizations (i.e., kernels that can extract a high-level of parallelism in the FPGA). We also note that the HLS-V Only scenario achieves gains in performance higher than in energy. It occurs since performance gains comes with the cost of increasing power dissipation, created by a higher FPGA resource usage required from the optimized kernel versions. Overall, *HLS-Versioning* provided 3.367x EDP improvements, outperforming the 3.276x gain provided by selecting the same HLS version for all FPGA nodes (Single Configuration), as shown in Section 5.4.1.

Considering both *Thread Throttling* and *HLS-Versioning*, the CPU-TT + HLS-V scenario provided the greatest benefits in performance (7.721x) and energy consumption (5.366x), resulting in a 41.434x EDP improvement. Such substantial improvements came from the synergistic effect of optimizing both CPU and FPGA kernels, exemplified in Figure 5.6, since optimizing only at one device can limit the gains because of bottlenecks caused by disproportionately long execution time of non-optimized kernels.

As a representative example, Table 5.5 details the first ten versions selected in the CPU-TT + HLS-V scenario in 4 distinct DAGs. As already mentioned, those were the versions that presented the best EDP for each individual node. We can point out that for both CPU and FPGA versions are quite varied, ranging from 1 to 64 threads and HLS version from 4 to 6, reinforcing the benefits of the multiple configuration approach. These experiments show us that DAGs benefit from having different individual configurations, as they comprise kernels with variant behavior (e.g., level of FPGA acceleration or optimal #Threads) that will select different #Threads and different HLS versions.

Figure 5.7: Improvements provided by multiple configuration scenarios.



Source: the author

Table 5.5: Versioning Variation for CPU/FPGA kernels of t threads or HLS version v.

	First N nodes									
DAG	1	2	3	4	5	6	7	8	9	10
1	v6	24t	1t	v6	v6	64t	15t	v6	4t	v5
2	3t	v6	v4	3t	v6	64t	v6	64t	15t	v4
3	56t	v4	v4	4t	v5	24t	v6	4t	15t	v6
4	v6	1t	4t	v6	v5	3t	64t	v6	v5	64t

Source: the author

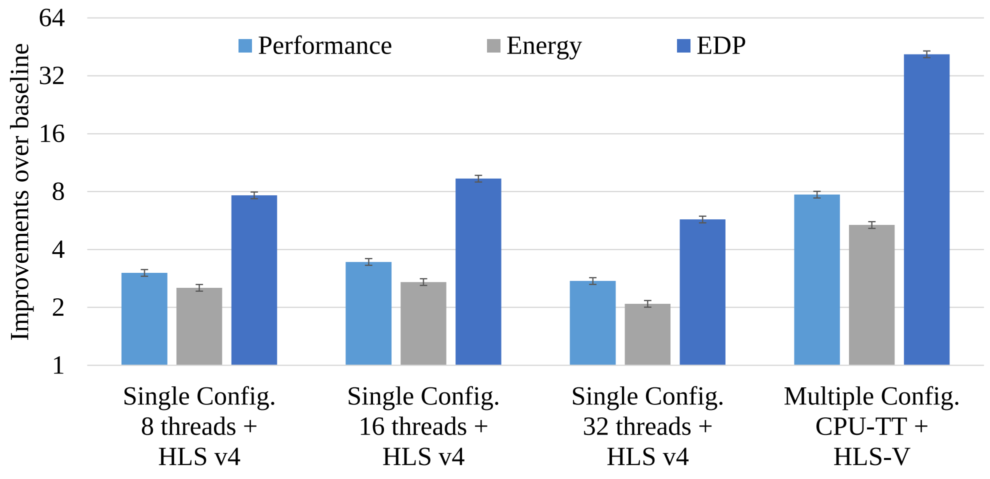
5.4.3 Single vs Multiple Configuration

Finally, we compared both single and multiple configuration approaches, in order to quantify the benefits of individually tuning the nodes (multiple configurations) in contrast to setting a fixed configurations for all nodes (single configuration).

Figure 5.8 shows Performance, Energy, and EDP improvements over the baseline, provided by the top 3 single configuration combinations compared to the best multiple configuration approach (CPU-TT+HLS-V), which applies both *Thread Throttling* and *HLS-Versioning* techniques. As previously pointed out, single configuration with 16 threads + HLS v4 provided the biggest gains, with 9.348x EDP gains, but was still outpaced over 4x by the Multiple Configuration CPU-TT+HLS-V approach, which yielded 41.434x EDP gains.

Single Configuration solutions have the clear advantage of its easy employment. However, even when the approach reaches a sweet spot (in our experiments: 16 threads + HLS v4), it is not capable of outperforming the fine tuning of Multiple Configurations, which demonstrably enables improvements in kernels, which the optimal #Threads and

Figure 5.8: Improvements from Single and Multiple Configuration scenarios.



Source: the author

HLS version, differ from the average value of Single Configuration. Consequently, the Multiple Configurations proved to be the most robust approach to workload variations.

Our findings demonstrate that single configuration can provide up to 9.348x EDP gains over the baseline, which considers the system's maximum number of threads and no-use of HLS pragmas. Instead, applying *Thread Throttling* and *HLS-Versioning* using multiple configurations for each kernel increases the EDP improvements to 41.434x, while being more robust to workload changes.

In the end, to fully exploit the benefits of *Thread Throttling* and *HLS-Versioning*, a solution that takes the global DAG execution is required. However, given the scalability issues shown in Section 5.3, heuristics like the one proposed in this work are mandatory to reach such a solution efficiently.

6 CONCLUSION

In this MSc dissertation, we investigated means to optimize the performance and energy consumption of emerging Collaborative Computing in heterogeneous systems. Such environments face the challenge of extracting all the benefits offered by such systems, once their devices present variable efficiency, preventing the programmers from statically reaching optimal execution.

This work investigated, in Chapter 3, the impact of the number of CPU threads and the workload balance of CPU-GPU and CPU-FPGA applications, where experimental evaluation presented considerable exploration space available with both strategies. Findings showed that the optimal number of threads depends on the application, the workload balancing among the devices, and the optimization target metric (performance, energy, or EDP). Also, to achieve the best results, CPU-FPGA applications are more dependent on the workload balance than the CPU-GPU ones, since some applications better suit one of the devices, so naively workload balancing stands far from optimal results.

To tackle the optimization opportunities of collaborative environments, we proposed ETCG and ETCF, which comprise approaches capable of improving performance and energy consumption of CPU-GPU and CPU-FPGA systems, respectively. ETCG comprises a the C++ thread library extension to enable automatic CPU *Thread Throttling* in CPU-GPU collaborative applications. It determines a near-optimal number of CPU threads using a Hill-Climbing based algorithm, which can rapidly converge to the solution, avoiding a prohibitive exhaustive search for such run-time approach. Compared to the static use of the maximum number of threads available, ETCG improves performance in 2.56x, while reducing EDP by 52% on average, with up to 73% on the best case. ETCG could also provide near-optimal results, standing a performance improvement only 6% below from the offline exhaustive search (Oracle). Considering CPU-FPGA collaborative applications, we proposed ETCF framework, which optimizes both number of CPU threads and workload balance. The framework comprises the C++ thread library extension proposed on ETCG and a workload balance approach using OpenCL. It provides the administrator the selection of each optimization and a optimization goal. Compared to the static use of maximum number of CPU threads and evenly workload balancing, ETCF increases performance by 6.67x, while reducing the energy consumption and EDP by 78% and 93%, respectively.

The work also carried out an investigation on the benefits on optimizations on task-collaborative environment, using a CPU-FPGA multi-tenant cloud environment as object of study. Specifically, we employed *Thread Throttling* to the CPU kernels and *HLS-Versioning* to the FPGA kernels. The resource provisioning restrictions from such environments demand a collaboratively use of the heterogeneous system, so both optimization techniques must be employed together. We modeled our experiments considering a DAG-alike execution arrangement, which is characteristic from the multi-tenant environments. Hence, we elaborated two approaches to optimize the execution: using a single configuration (number of CPU threads and HLS version) for all kernels in the DAG; and using multiple configurations individually tuned for each kernel. Compared to using the maximum number of CPU threads and non-optimized FPGA execution, single configuration experiments provide 9.4x EDP improvements, while multiple configurations boosted the EDP gains to 41.4x.

6.1 Future Work

The studies developed in this work opened up other potential research opportunities. The ETCG approach can be implemented and improved using threading APIs other than C++11 and OpenMP, which could extend the number of supported applications. Another possible work is searching for a way of implementing a fully online version of the ETCF framework, which could perform both *Thread Throttling* and workload balance at run-time. For that, it will be required modifications on the OpenCL API to support the dynamic adjustment of the data partitioning of the applications. Additionally, the final study developed in this dissertation is likely the one with the most straightforward attainable future work, once this work stick to the DSE of the multi-tenant CPU-FPGA cloud environment using an effortless heuristic to show the room for improvements available. In this scenario, other heuristics may be considered taking the whole DAG execution into account when evaluating which configuration to select at each kernel.

6.2 Publications

As a result of the work developed on the course of the Master's program, the following publications have been made.

- KNORST, T. et al. Etcg: Energy-aware cpu thread throttling for cpu-gpu collaborative environments. In: IEEE. **2021 34th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.], 2021. p. 1–6.
- KNORST, T. et al. Etcf – energy-aware cpu thread throttling and workload balancing framework for cpu-fpga collaborative environments. In: IEEE. **2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC)**. [S.l.], 2021. p. 1–8.
- KNORST, T. et al. On the benefits of collaborative thread throttling and hls-versioning in cpu-fpga environments. In: IEEE. **2022 35th SBC/SBMicro/IEEE/ ACM Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.], 2022. p. 1–6.

In addition, the author collaborated as a Master's student in the following works:

- VICENZI, J. C. et al. Tripp: Transparent resource provisioning for multi-tenant cpu-gpu based cloud environments. In: IEEE. **2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC)**. [S.l.], 2021. p. 1–8.
- JORDAN, M. G. et al. Erin: Energy-aware resource provisioning framework for cpu-fpga multi-tenant environment. **IEEE Design & Test**, IEEE, 2022.

REFERENCES

BECK, A. C. S.; LISBÔA, C. A. L.; CARRO, L. **Adaptable embedded systems**. [S.l.]: Springer Science & Business Media, 2012.

CHE, S. et al. Rodinia: A benchmark suite for heterogeneous computing. In: IEEE. **2009 IEEE international symposium on workload characterization (IISWC)**. [S.l.], 2009. p. 44–54.

CHEN, F. et al. Enabling fpgas in the cloud. In: **Proceedings of the 11th ACM Conference on Computing Frontiers**. [S.l.: s.n.], 2014. p. 1–10.

CHOI, Y.-k.; CONG, J. Hls-based optimization and design space exploration for applications with variable loop bounds. In: IEEE. **2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.], 2018. p. 1–8.

CONG, J. et al. Cpu-fpga coscheduling for big data applications. **IEEE Design & Test**, IEEE, v. 35, n. 1, p. 16–22, 2017.

CONG, J. et al. Understanding performance differences of fpgas and gpus. In: IEEE. **2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)**. [S.l.], 2018. p. 93–96.

CRAVEN, S.; ATHANAS, P. Examining the viability of fpga supercomputing. **EURASIP Journal on Embedded systems**, Springer, v. 2007, p. 1–8, 2007.

CURTIS-MAURY, M. et al. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In: **Proceedings of the 20th annual international conference on Supercomputing**. [S.l.: s.n.], 2006. p. 157–166.

CURTIS-MAURY, M. et al. Prediction models for multi-dimensional power-performance optimization on many cores. In: **Proceedings of the 17th international conference on Parallel architectures and compilation techniques**. [S.l.: s.n.], 2008. p. 250–259.

DAGUM, L.; MENON, R. Openmp: an industry standard api for shared-memory programming. **IEEE computational science and engineering**, IEEE, v. 5, n. 1, p. 46–55, 1998.

DEIANA, E. A. et al. A multiobjective reconfiguration-aware scheduler for fpga-based heterogeneous architectures. In: IEEE. **2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)**. [S.l.], 2015. p. 1–6.

DU, P. et al. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. **Parallel Computing**, Elsevier, v. 38, n. 8, p. 391–407, 2012.

FAHMY, S. A.; VIPIN, K.; SHREEJITH, S. Virtualized fpga accelerators for efficient cloud computing. In: IEEE. **2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)**. [S.l.], 2015. p. 430–435.

GÓMEZ-LUNA, J. et al. Chai: Collaborative heterogeneous applications for integrated-architectures. In: IEEE. **2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.], 2017. p. 43–54.

HACKENBERG, D. et al. Power measurement techniques on standard compute nodes: A quantitative comparison. In: IEEE. **2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.], 2013. p. 194–204.

HÄHNEL, M. et al. Measuring energy consumption for short code paths using rapl. **ACM SIGMETRICS Performance Evaluation Review**, ACM New York, NY, USA, v. 40, n. 3, p. 13–17, 2012.

HILMAN, M. H.; RODRIGUEZ, M. A.; BUYYA, R. Multiple workflows scheduling in multi-tenant distributed systems: A taxonomy and future directions. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 53, n. 1, p. 1–39, 2020.

HUANG, S. et al. Analysis and modeling of collaborative execution strategies for heterogeneous cpu-fpga architectures. In: **Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering**. [S.l.: s.n.], 2019. p. 79–90.

ILIĆ, A.; SOUSA, L. Collaborative execution environment for heterogeneous parallel systems. In: IEEE. **2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)**. [S.l.], 2010. p. 1–8.

ISO, I. Iec 14882: 2011 information technology—programming languages—c++. **International Organization for Standardization, Geneva, Switzerland**, v. 27, p. 59, 2012.

JOHNSON, A. W.; JACOBSON, S. H. On the convergence of generalized hill climbing algorithms. **Discrete applied mathematics**, Elsevier, v. 119, n. 1-2, p. 37–57, 2002.

JORDAN, M. G. et al. Erin: Energy-aware resource provisioning framework for cpu-fpga multi-tenant environment. **IEEE Design & Test**, IEEE, 2022.

JORDAN, M. G. et al. Resource-aware collaborative allocation for cpu-fpga cloud environments. **IEEE Transactions on Circuits and Systems II: Express Briefs**, IEEE, v. 68, n. 5, p. 1655–1659, 2021.

KACHRIS, C.; SOUDRIS, D. A survey on reconfigurable accelerators for cloud computing. In: IEEE. **2016 26th International conference on field programmable logic and applications (FPL)**. [S.l.], 2016. p. 1–10.

KNORST, T. et al. Etcg: Energy-aware cpu thread throttling for cpu-gpu collaborative environments. In: IEEE. **2021 34th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.], 2021. p. 1–6.

KNORST, T. et al. Etcf – energy-aware cpu thread throttling and workload balancing framework for cpu-fpga collaborative environments. In: IEEE. **2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC)**. [S.l.], 2021. p. 1–8.

KNORST, T. et al. On the benefits of collaborative thread throttling and hls-versioning in cpu-fpga environments. In: IEEE. **2022 35th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI)**. [S.l.], 2022. p. 1–6.

LEE, J. et al. Skmd: Single kernel on multiple devices for transparent cpu-gpu collaboration. **ACM Transactions on Computer Systems (TOCS)**, ACM New York, NY, USA, v. 33, n. 3, p. 1–27, 2015.

LEE, J. et al. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In: **Proceedings of the 37th annual international symposium on Computer architecture**. [S.l.: s.n.], 2010. p. 270–279.

LIGNATI, B. N. et al. Exploiting hls-generated multi-version kernels to improve cpu-fpga cloud systems. In: IEEE. **2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)**. [S.l.], 2021. p. 536–541.

LORENZON, A. F.; FILHO, A. C. S. B. **Parallel Computing Hits the Power Wall: Principles, Challenges, and a Survey of Solutions**. [S.l.]: Springer Nature, 2019.

LORENZON, A. F. et al. Aurora: Seamless optimization of openmp applications. **IEEE transactions on parallel and distributed systems**, IEEE, v. 30, n. 5, p. 1007–1021, 2018.

LORENZON, A. F.; SOUZA, J. D.; BECK, A. C. S. Laant: A library to automatically optimize edp for openmp applications. In: IEEE. **Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017**. [S.l.], 2017. p. 1229–1232.

MARATHE, A. et al. A run-time system for power-constrained hpc applications. In: SPRINGER. **International conference on high performance computing**. [S.l.], 2015. p. 394–408.

MELONI, P. et al. Neuraghe: Exploiting cpu-fpga synergies for efficient and flexible cnn inference acceleration on zynq socs. **ACM Transactions on Reconfigurable Technology and Systems (TRETS)**, ACM New York, NY, USA, v. 11, n. 3, p. 1–24, 2018.

PHAM, N. K. et al. Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis. In: IEEE. **2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.], 2015. p. 157–162.

PORTERFIELD, A. K. et al. Power measurement and concurrency throttling for energy reduction in openmp programs. In: IEEE. **2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum**. [S.l.], 2013. p. 884–891.

PUSUKURI, K. K.; GUPTA, R.; BHUYAN, L. N. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In: IEEE. **2011 IEEE International Symposium on Workload Characterization (IISWC)**. [S.l.], 2011. p. 116–125.

SENSI, D. D.; TORQUATI, M.; DANELUTTO, M. A reconfiguration algorithm for power-aware parallel applications. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM New York, NY, USA, v. 13, n. 4, p. 1–25, 2016.

SHAFIK, R. A. et al. Adaptive energy minimization of openmp parallel applications on many-core systems. In: **Proceedings of the 6th Workshop on Parallel Programming**

and Run-Time Management Techniques for Many-core Architectures. [S.l.: s.n.], 2015. p. 19–24.

STONE, J. E.; GOHARA, D.; SHI, G. Opencl: A parallel programming standard for heterogeneous computing systems. **Computing in science & engineering**, IEEE Computer Society, v. 12, n. 3, p. 66, 2010.

SULEMAN, M. A.; QURESHI, M. K.; PATT, Y. N. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps. **ACM Sigplan Notices**, ACM New York, NY, USA, v. 43, n. 3, p. 277–286, 2008.

TAKACH, A. High-level synthesis: Status, trends, and future directions. **IEEE Design & Test**, IEEE, v. 33, n. 3, p. 116–124, 2016.

VICENZI, J. C. et al. Tripp: Transparent resource provisioning for multi-tenant cpu-gpu based cloud environments. In: IEEE. **2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC)**. [S.l.], 2021. p. 1–8.

WANG, S.; ANANTHANARAYANAN, G.; MITRA, T. Optic: Optimizing collaborative cpu–gpu computing on mobile devices with thermal constraints. **IEEE transactions on computer-aided design of integrated circuits and systems**, IEEE, v. 38, n. 3, p. 393–406, 2018.

WEI, X. et al. Throughput optimization for streaming applications on cpu-fpga heterogeneous systems. In: IEEE. **2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)**. [S.l.], 2017. p. 488–493.

WU, J. et al. Cloud storage as the infrastructure of cloud computing. In: IEEE. **2010 International conference on intelligent computing and cognitive informatics**. [S.l.], 2010. p. 380–383.

ZENG, H.; PRASANNA, V. Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms. In: **Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. [S.l.: s.n.], 2020. p. 255–265.

ZHAO, J. et al. Performance modeling and directives optimization for high-level synthesis on fpga. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 39, n. 7, p. 1428–1441, 2019.

ZHONG, G. et al. Lin-analyzer: A high-level performance analysis tool for fpga-based accelerators. In: IEEE. **2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)**. [S.l.], 2016. p. 1–6.

ZHOU, S.; PRASANNA, V. K. Accelerating graph analytics on cpu-fpga heterogeneous platform. In: IEEE. **2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)**. [S.l.], 2017. p. 137–144.

APPENDIX A — ETCG EXPERIMENTS

Table A.1: Execution time (s) of the applications at each scenario using α=0.3

Benchmark	Max #Threads	Bench. Default	ETCG	Oracle
BS	5.581	7.393	2.863	2.856
CED-D	5.940	5.481	5.344	5.204
HSTI	55.087	12.965	7.938	7.938
HSTO	13.472	9.033	7.328	7.310
PAD	7.327	3.207	2.710	2.631
RSC-D	27.668	42.772	11.853	11.425
SC	6.485	2.496	1.872	1.867

Source: the author

Table A.2: Execution time (s) of the applications at each scenario using α=0.5

Benchmark	Max #Threads	Bench. Default	ETCG	Oracle
BS	5.662	12.050	2.758	2.676
CED-D	6.307	5.773	5.474	5.335
HSTI	52.602	16.999	9.942	9.942
HSTO	13.063	10.666	7.490	7.394
PAD	7.494	4.041	2.855	2.855
RSC-D	28.528	73.650	14.310	14.310
SC	6.813	3.456	2.186	2.186

Source: the author

Table A.3: Execution time (s) of the applications at each scenario using α=0.7

Benchmark	Max #Threads	Bench. Default	ETCG	Oracle
BS	5.746	16.493	3.171	3.171
CED-D	6.817	6.018	5.497	5.497
HSTI	54.749	22.914	11.513	11.513
HSTO	12.986	11.239	7.483	7.483
PAD	7.836	4.665	2.973	2.973
RSC-D	28.575	93.559	17.404	16.929
SC	7.103	4.399	2.417	2.417

Source: the author

Table A.4: Energy consumption (J) of the applications at each scenario using α=0.3

Benchmark	Max #Threads	Bench. Default	ETCG	Oracle
BS	913.1	1037.6	520.5	520.5
CED-D	699.7	634.4	619.6	604.2
HSTI	6800.0	1618.3	1039.9	1039.9
HSTO	2903.4	1141.4	1060.6	1060.6
PAD	924.4	400.3	345.5	345.5
RSC-D	3851.4	5337.9	1925.4	1865.2
SC	837.2	322.0	292.9	292.9

Source: the author

Table A.5: Energy consumption (J) of the applications at each scenario using α=0.5

Benchmark	Max #Threads	Bench. Default	ETCG	Oracle
BS	906.8	1509.3	559.4	552.6
CED-D	747.5	680.2	635.9	625.0
HSTI	6533.7	2106.4	1297.8	1297.8
HSTO	3030.1	1310.4	1145.4	1117.1
PAD	937.3	502.1	388.9	387.3
RSC-D	4232.6	9085.0	2399.2	2399.2
SC	861.8	431.9	300.9	300.9

Source: the author

Table A.6: Energy consumption (J) of the applications at each scenario using α=0.7

Benchmark	Max #Threads	Bench. Default	ETCG	Oracle
BS	898.6	2046.8	641.0	637.5
CED-D	799.6	699.3	646.1	646.1
HSTI	6828.3	2832.1	1535.2	1518.3
HSTO	2994.5	1390.0	1200.3	1116.7
PAD	986.9	576.6	410.6	410.6
RSC-D	4546.0	11440.7	2977.3	2977.3
SC	904.1	555.9	339.3	339.3

Source: the author

APPENDIX B — ETCF EXPERIMENTS

Table B.1: Execution time (s) of the applications at each scenario.

App.	Max #Threads	App. Default	ETCF TT	ETCF WB	ETCF Full	Oracle
BS	1.771	1.766	1.754	0.561	0.561	0.549
CED-D	12.779	12.758	12.779	7.022	5.748	5.593
HSTI	0.454	0.31	0.262	0.45	0.231	0.23
HSTO	0.91	0.764	0.618	0.881	0.515	0.509
PAD	0.788	0.782	0.778	0.586	0.37	0.358
RSC-D	8.314	8.307	8.289	1.994	1.994	1.971
SC	0.53	0.382	0.379	0.502	0.33	0.322

Source: the author

Table B.2: Energy consumption (J) of the applications at each scenario.

App.	Max #Threads	App. Default	ETCF TT	ETCF WB	ETCF Full	Oracle
BS	183.3	167.5	181.6	67.6	65.7	63.8
CED-D	1419.2	1408.5	1413.7	804.4	1158.4	1127.2
HSTI	41.8	23.3	22.7	40.1	19.3	19.3
HSTO	167.1	90.5	99.7	145	78.1	72.3
PAD	81.6	80.9	80	57.2	40.1	38.9
RSC-D	773.5	758.4	752.5	211.9	210.2	190.6
SC	50.1	36.2	37.2	49.1	35.3	34

Source: the author

APPENDIX C — CHAPTER 5 EXPERIMENTS

Table C.1: Single Configuration results - 1 to 16 threads / HLS v1 to v6

Single Config. #Threads	Single Config. HLS Ver.	Average Execution Time (s)	Std. Dev.	Average Energy Cons. (kJ)	Std. Dev.	Average System Power (W)
1	1	3728.65	±4.27%	255.91	±4.31%	68.63
1	2	3727.41	±4.28%	255.51	±4.31%	68.55
1	3	3727.40	±4.28%	255.43	±4.31%	68.53
1	4	3727.40	±4.28%	255.34	±4.31%	68.51
1	5	3727.40	±4.28%	255.37	±4.31%	68.51
1	6	3727.40	±4.28%	255.41	±4.31%	68.52
2	1	2077.61	±4.00%	145.89	±3.98%	70.22
2	2	2066.65	±4.14%	145.12	±4.19%	70.22
2	3	2066.64	±4.14%	145.04	±4.19%	70.18
2	4	2066.64	±4.14%	144.96	±4.19%	70.14
2	5	2066.64	±4.14%	144.97	±4.19%	70.15
2	6	2066.64	±4.14%	145.01	±4.19%	70.17
4	1	2006.25	±3.68%	118.98	±3.89%	59.30
4	2	1194.88	±4.00%	87.39	±4.05%	73.14
4	3	1194.86	±4.00%	87.31	±4.05%	73.07
4	4	1194.85	±4.01%	87.23	±4.05%	73.01
4	5	1194.85	±4.01%	87.25	±4.05%	73.02
4	6	1194.85	±4.00%	87.29	±4.05%	73.05
8	1	2005.92	±3.72%	103.56	±3.89%	51.63
8	2	663.13	±3.99%	51.51	±3.99%	77.68
8	3	663.10	±3.99%	51.43	±3.99%	77.56
8	4	663.10	±3.99%	51.36	±3.99%	77.45
8	5	663.10	±3.99%	51.37	±3.99%	77.47
8	6	663.10	±3.99%	51.41	±3.99%	77.53
16	1	2005.89	±3.71%	103.29	±3.89%	51.49
16	2	581.63	±3.79%	48.11	±3.84%	82.71
16	3	581.60	±3.79%	48.02	±3.84%	82.57
16	4	581.60	±3.79%	47.95	±3.84%	82.44
16	5	581.60	±3.79%	47.96	±3.84%	82.46
16	6	581.60	±3.79%	48.00	±3.84%	82.54

Source: the author

Table C.2: Single Configuration results - 1 to 16 threads / HLS v1 to v6 - and Multiple Configurations at the end.

Single Config. #Threads	Single Config. HLS Ver.	Average Execution Time (s)	Std. Dev.	Average Energy Cons. (kJ)	Std. Dev.	Average System Power (W)
32	1	2005.91	±3.67%	111.80	±3.89%	55.74
32	2	730.98	±3.80%	62.37	±3.93%	85.32
32	3	730.95	±3.80%	62.29	±3.93%	85.22
32	4	730.94	±3.80%	62.21	±3.93%	85.12
32	5	730.94	±3.80%	62.23	±3.93%	85.13
32	6	730.94	±3.80%	62.27	±3.93%	85.19
64	1	2005.99	±3.70%	128.86	±3.89%	64.24
64	2	903.91	±4.02%	86.08	±4.05%	95.23
64	3	903.87	±4.02%	86.00	±4.05%	95.14
64	4	903.86	±4.02%	85.92	±4.05%	95.06
64	5	903.86	±4.02%	85.93	±4.05%	95.07
64	6	903.86	±4.02%	85.97	±4.05%	95.12
128	1	2005.99	±3.70%	129.96	±3.89%	64.79
128	2	911.47	±4.01%	87.47	±4.04%	95.97
128	3	911.43	±4.01%	87.39	±4.04%	95.89
128	4	911.42	±4.02%	87.32	±4.04%	95.80
128	5	911.42	±4.02%	87.33	±4.04%	95.82
128	6	911.42	±4.01%	87.37	±4.04%	95.86
Mult. Config. CPU-TT Only		1999.17	±3.74%	91.65	±3.90%	45.85
Mult. Config. HLS-V Only		897.14	±4.06%	86.32	±4.10%	96.21
Mult. Config. CPU-TT + HLS-V		259.78	±3.58%	24.22	±3.84%	93.23

Source: the author