PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

RENATO BARRETO HOFFMANN FILHO

IMPACTS OF PARALLEL PROGRAMMING ON LIMITED-RESOURCE HARDWARE

Porto Alegre
2023

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

IMPACTS OF PARALLEL PROGRAMMING ON LIMITED-RESOURCE HARDWARE

RENATO BARRETO HOFFMANN FILHO

Master Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Dr. Dalvan Jair Griebler
Co-Advisor: Prof. Dr. Luiz Gustavo Leão Fernandes

Porto Alegre
2023

Ficha Catalográfica

H711i Hoffmann Filho, Renato Barreto

Impacts of parallel programming on limited-resource hardware / Renato Barreto Hoffmann Filho. – 2023.
103 f.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Dalvan Jair Griebler.
Coorientador: Prof. Dr. Luiz Gustavo Leão Fernandes.

1. parallelism. 2. benchmarking. 3. energy consumption. 4. limited resource hardware. 5. embedded systems. I. Griebler, Dalvan Jair. II. Leão Fernandes, Luiz Gustavo. III. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

RENATO BARRETO HOFFMANN FILHO

IMPACTS OF PARALLEL PROGRAMMING ON LIMITED-RESOURCE HARDWARE

This Master Thesis has been submitted in partial fulfillment of the requirements for the degree of Master in Computer Science, of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on March 30th, 2023.

COMMITTEE MEMBERS:

Prof. Dr. Patrizio Dazzi (University of Pisa)

Prof. Dr. César Augusto Mission Marcon (PPGCC/PUCRS)

Prof. Dr. Luiz Gustavo Leão Fernandes (PPGCC/PUCRS- Co-Advisor)

Prof. Dr. Dalvan Jair Griebler (PPGCC/PUCRS - Advisor)

ACKNOWLEDGMENTS

I want to thank many people that contributed to this work. First, I would like to thank my advisors for their valuable guidance. Second, my peers and colleagues in our research group for the valuable feedback and discussions and for helping me on many occasions. Third, PUCRS University, its professors, and all its collaborators during my formative years. This research is partially funded by MCTIC/CNPq call 25/2020 (Nº 130484/2021-0) and SAP enterprise.

IMPACTS OF PARALLEL PROGRAMMING ON LIMITED-RESOURCE HARDWARE

ABSTRACT

Limited resource hardware devices are more affordable and energy efficient than high-end hardware. Despite their reduced size, these devices are increasingly complex, with many now featuring multiple processing cores, GPGPU accelerators, and larger RAM capacity. To fully utilize their computational capacity, software developers must exploit parallelism, but this adds an extra layer of complexity because they must balance computational constraints and performance demands. Therefore, choosing the appropriate parallelism strategy and parallel programming interface is crucial to achieving the best hardware performance. To tackle this problem, we defined research objectives to guide our work in finding the most appropriate parallelism strategies and programming interfaces for limited-resource hardware regarding performance and energy consumption. We experimented with 12 applications using three devices and seven parallel programming interfaces. This thesis introduces new metrics, additional applications, various parallelism interfaces, and extra hardware devices. We developed a structured set of research objectives to evaluate parallelism, providing a methodology to organize many parallelism considerations. In summary, this study concludes that parallel computing is beneficial in limited-resource hardware, and higher-level of abstraction parallel programming interfaces are viable options. Our results on target architecture and specific parallelism models indicate that parallelism benefits limited-resource hardware, reducing total energy consumption by up to 63.53% and increasing throughput by up to 112.54%. Additionally, power peak differences are up to 24.98% between programming techniques. Another indication is that there are estimated software complexity differences between programming interfaces of up to 858.33%. Overall, this thesis contributes to understanding the impacts of parallel programming on limited-resource hardware and provides insights into optimizing parallel programs for such hardware. Our findings can be helpful for researchers, developers, and engineers working on parallel programming for limited-resource hardware.

Keywords: limited resource hardware, parallelism, energy consumption, embedded systems, benchmarking.

IMPACTOS DA PROGRAMAÇÃO PARALELA EM DISPOSITIVOS COM RECURSOS LIMITADOS

RESUMO

Dispositivos de hardware com recursos limitados são mais acessíveis e energeticamente eficientes do que hardware de ponta. Apesar de seu tamanho reduzido, esses dispositivos estão cada vez mais complexos, muitos agora apresentando vários núcleos de processamento, aceleradores GPGPU e maior capacidade de RAM. Para aproveitar ao máximo sua capacidade computacional, os desenvolvedores de software devem explorar o paralelismo, mas isso adiciona uma camada extra de complexidade, pois eles devem lidar com as restrições computacionais e as demandas de desempenho. Portanto, escolher a estratégia de paralelismo apropriada e a interface de programação paralela é crucial para obter o melhor desempenho do hardware. Para enfrentar esse problema, foram definidos objetivos de pesquisa para orientar a pesquisa sobre as estratégias de paralelismo e interfaces de programação mais adequadas para hardware com recursos limitados em relação ao desempenho e consumo de energia. Foram realizados experimentos com 12 aplicações usando três dispositivos e sete interfaces de programação paralela. Esta tese apresentpu novas métricas, diferentes aplicações, várias interfaces de paralelismo e diferentes dispositivos de hardware. Foi desenvolvido um conjunto estruturado de objetivos de pesquisa para avaliar o paralelismo, fornecendo uma metodologia para organizar várias considerações de paralelismo. Em resumo, este estudo concluiu que a computação paralela é benéfica em hardware com recursos limitados. Além disso, interfaces de programação paralela de nível mais alto de abstração são opções viáveis. Os resultados em dispositivos e interfaces específicas indicaram que o paralelismo beneficia o hardware com recursos limitados, reduzindo o consumo total de energia em até 63,53% e a vazão em até 112,54%. Além disso, as diferenças de pico de energia são de até 24,98% entre as técnicas de programação. Outra indicação é que existem diferenças estimadas de complexidade de software entre as interfaces de programação de até 858,33%. Em geral, esta tese contribuiu para a compreensão dos impactos da programação paralela em hardware com recursos limitados e fornece *insights* para otimizar programas paralelos para esse hardware. Nossas descobertas podem ser úteis para pesquisadores, desenvolvedores e engenheiros que trabalham com programação paralela para hardware com recursos limitados.

Palavras-Chave: recursos de hardware limitados, paralelismo, consumo de energia, sistemas embarcados, testes.

LIST OF FIGURES

Figure 2.1 – MapReduce algorithm. Vector addition example. 20

Figure 2.2 – High-Level SPar Usability. 29

Figure 2.3 – SPar language Face Recognition example. 29

Figure 2.4 – SPar language attributes example. 30

Figure 2.5 – SPar compilation process flow. 32

Figure 2.6 – Parallel Face Recognizer visualization of the necessary code with FastFlow. 34

Figure 2.7 – Parallel Face Recognizer visualization of the necessary code with TBB. 34

Figure 2.8 – Parallel Face Recognizer visualization of the necessary code with OpenMP. 35

Figure 2.9 – Parallel Face Recognizer visualization of the necessary code with Threads. 36

Figure 3.1 – Search string on Scopus scientific database. 37

Figure 4.1 – Stream processing applications execution flow. 45

Figure 4.2 – Data processing applications execution flow. 47

Figure 4.3 – MapReduce. 50

Figure 5.1 – Experimental methodology using the research objectives. 54

Figure 5.2 – SPBench applications SLOC. 57

Figure 5.3 – NPB applications SLOC. 58

Figure 5.4 – EP application on multiple devices. 62

Figure 5.5 – FT on multiple devices. 63

Figure 5.6 – MG on multiple devices. 64

Figure 5.7 – CG on multiple devices. 65

Figure 5.8 – LU on multiple devices. 66

Figure 5.9 – BT on multiple devices. 67

Figure 5.10 – SP on multiple devices. 68

Figure 5.11 – Odroid proof for a correct thread-to-core mapping. 69

Figure 5.12 – Raspberry device. 70

Figure 5.13 – correlation between execution time and energy consumption on data processing applications. 72

Figure 5.14 – Example of the problem in SPar code generation. 77

Figure 5.15 – Odroid device graphs. 78

Figure 5.16 – Ferret application: correlation between performance and energy consumption. 82

LIST OF TABLES

Table 3.1 – Criteria for selecting documents in the Literature Review process. . . . 37
Table 3.2 – Comparing related work with this Master’s thesis research. 41
Table 4.1 – Low-resource hardware configurations and kernel version. 43
Table 5.1 – Classification of Research Objectives. 53
Table 5.2 – Parallel Programming Interfaces main characteristics comparison. . . 55
Table 5.3 – Normalized Halstead for SPBench applications. 59
Table 5.4 – Normalized Halstead for NPB applications. Normalized values. 59
Table 5.5 – Execution time for NPB applications. For parallel versions, green and red high- light the best and worst observed results, respectively. 61
Table 5.6 – Energy consumption for NPB. The maximum observed standard deviation is 0.034. For parallel versions, green and red highlight the best and worst observed results respectively. 73
Table 5.7 – Power dissipation for NPB. The maximum observed standard deviation is 0.61. For parallel versions, green highlights the best and red the worst results. 75
Table 5.8 – Latency measurements for all SPBench applications. The standard deviation can vary between 0.0003 and 0.45. For parallel versions, green and red highlight the best and worst observed results respectively. 77
Table 5.9 – throughput of all SPBench applications. The maximum observed standard deviation is 0.92. For parallel versions, green and red highlight the best and worst observed results respectively. 79
Table 5.10 – Energy consumption of all SPBench applications. The maximum observed standard deviation is 0.00089. For parallel versions, green and red highlight the best and worst observed results respectively. 82
Table 5.11 – Power dissipation of all SPBench applications. The maximum observed standard deviation is 0.31. For parallel versions, green and red highlight the best and worst observed results respectively. 83
Table 5.12 – Memory consumption of all SPBench applications. For parallel versions, green and red highlight the best and worst observed results respectively. 84
Table 5.13 – Matrix Multiplication Metrics. The greatest standard deviations are 3.58 for the CPU and 2.21 for GPU. 85
Table 5.14 – Metrics for NPB applications with CUDA GPGPU. 87

LIST OF ACRONYMS

API – Application Programming Interface

AST – Abstract Syntax Tree

BT – Block Tri-diagonal solver

BZ2 – Bzip2

CG – Conjugate Gradient

CPU – Central Processing Unit

CUDA – Compute Unified Device Architecture

CINCLE – Compiler Infrastructure for new C/C++ Languages Extensions

DSP – Digital Signal Processing

DSL – Domain-Specific Language

EP – Embarrassingly Parallel

FE – Ferret

FR – Face Recognizer

FPGA – Field-Programmable Gate Array

FT – Fast Fourier Transform

GCC – Gnu Compiler Collection

GMAP – Parallel Applications Modeling Group

GPGPU – General Purpose Graphics Processing Unit

ILP – Instruction Level Parallelism

IS – Integer Sort

ISO – International Organization for Standardization

LD – Lane Detection

LU – Lower-Upper Gauss-Seidel solver

MG – Multi Grid

MPI – Message Passing Interface

OS – Operational System

PPI – Parallel Programming Interface

RO – Research Objectives

SIMD – Single Instruction Multiple Data

SLOC – Source Lines of Code

SLO – Service Level Objective

SP – Scalar Penta-diagonal solver

SPAR – Stream Parallelism

STL – Standard Template Library

TBB – Threading Building Blocks

TLP – Thread Level Parallelism

UAV – Unmanned Aerial Vehicle

CONTENTS

1 INTRODUCTION . . . 16

2 BACKGROUND . . . 19

2.1 PARALLELISM . . . 19

2.1.1 PARALLEL PROGRAMMING INTERFACES . . . 20

2.2 STREAM PROCESSING . . . 22

2.3 EMBEDDED SYSTEMS . . . 23

2.3.1 OPTIMIZATIONS AND TECHNIQUES . . . 24

2.4 SPAR . . . 28

2.4.1 SPAR LANGUAGE . . . 28

2.4.2 SPAR COMPILER . . . 31

2.4.3 SPAR RESEARCH . . . 32

2.5 EXAMPLES OF PARALLEL PROGRAMMING . . . 33

2.5.1 FASTFLOW . . . 33

2.5.2 THREADING BUILDING BLOCKS . . . 33

2.5.3 OPENMP . . . 35

2.5.4 THREADS . . . 35

3 RELATED WORK . . . 37

3.1 RELATED RESEARCH . . . 38

3.2 COMPARISON WITH THIS WORK . . . 40

4 METHODOLOGY . . . 42

4.1 RESEARCH OBJECTIVES . . . 42

4.2 HARDWARE AND SOFTWARE ENVIRONMENT . . . 43

4.3 APPLICATIONS . . . 43

4.3.1 STREAM PROCESSING . . . 44

4.3.2 DATA PARALLELISM . . . 45

4.4 METRICS . . . 48

4.4.1 PERFORMANCE . . . 48

4.4.2 CONSUMPTION . . . 48

4.4.3 PROGRAMMING PRODUCTIVITY . . . 49

4.5 PARALLEL PROGRAMMING INTERFACES . . . 51

4.6 EXECUTION AND GRAPHS 51
4.6.1 INPUT DATA 52

5 RESEARCH OBJECTIVES' ANALYSIS 53

5.1 RO1: EVALUATING PROGRAMMING FEATURES FOR THE PARALLEL PROGRAMMING INTERFACES 54
5.2 RO2: EVALUATING PROGRAMMABILITY METRICS FOR THE PARALLEL PROGRAMMING INTERFACES 56
5.3 RO3: EVALUATING PERFORMANCE METRICS FOR DATA PROCESSING APPLICATIONS 59
5.3.1 RO3.1: ANALYSIS OF MAPREDUCE WITH NO DATA DEPENDENCIES 60
5.3.2 RO3.2: EVALUATING MAP WITH MEMORY CONTENTION 61
5.3.3 RO3.3: EVALUATING DATA LOCALITY ON MAPS 62
5.3.4 RO3.4: EVALUATING MAP WITH DATA DEPENDENCY 64
5.3.5 RO3.5: ANALYZING MEMORY BANDWIDTH AS A PARALLELISM BOTTLENECK . . . 65
5.3.6 RO3.6: EVALUATING BIG.LITTLE ARCHITECTURE PERFORMANCE FEATURES . . . 67
5.3.7 RO3.7: ANALYSIS OF THE IMPACT OF HARDWARE TEMPERATURE ON PERFORMANCE 68
5.4 RO4: EVALUATING CONSUMPTION METRICS FOR DATA PROCESSING APPLICATIONS 71
5.5 RO5: EVALUATING PERFORMANCE METRICS OF STREAM PROCESSING APPLICATIONS 74
5.5.1 RO5.1: ASSESSING THE EFFECT OF BLOCKING VS. NON-BLOCKING COMPUTATION ON PERFORMANCE METRICS 79
5.5.2 RO5.2: EVALUATING COMPUTATIONAL RESOURCE ALLOCATION STRATEGIES IN PARALLELISM TECHNIQUES 80
5.6 RO6: EVALUATING CONSUMPTION METRICS FOR STREAM PROCESSING APPLICATIONS 81
5.6.1 RO6.1: ANALYSIS OF MEMORY CONSUMPTION FOR PARALLEL STREAM APPLICATIONS 83
5.7 RO7: EVALUATING MATRIX MULTIPLICATION IN LIMITED-RESOURCE HARDWARE 84
5.8 RO8: EVALUATING GPGPU PARALLELISM FOR DATA PROCESSING APPLICATIONS 86

6 SUMMARY OF FINDINGS 88

7 CONCLUSION 92

REFERENCES 95

1. INTRODUCTION

Limited-resource hardware refers to systems or devices with constraints on processing capabilities, energy consumption, available memory, and storage. Firstly, they are more affordable and accessible than high-end hardware systems. Secondly, limited-resource hardware is more energy efficient [84, 35], leading to financial and environmental benefits in energy bill savings and lower carbon footprints. These devices are prevalent in embedded systems that perform specific functions such as mobile computing, industrial automation, and robotics [59]. In addition, there has also been a growing demand for more powerful and flexible embedded systems that can perform a broader range of functions, especially in the edge computing research area [49].

Limited-resource hardware is also an option for low-end general-purpose computational systems [35]. These devices have become increasingly sophisticated, with many now featuring multicore processors and accelerators such as GPGPUs. On the other hand, this evolution puts pressure on software developers; to use modern devices fully, developers must be proficient in parallel programming, possess good hardware knowledge, handle heterogeneous environments, write efficient and optimized code, and handle software portability [76].

Programming for limited-resource hardware typically involves lower-level and compiled languages such as C/C++ [13, 59]. Furthermore, selecting an appropriate PPI (parallel programming interface) adds an extra decision layer. Fortunately, there already exist several C++ solutions that offer suitable abstractions for this purpose [81, 60]. However, these interfaces have different design goals and implementation strategies. Some interfaces perform better than others under specific circumstances [54, 1]. Therefore, developers must understand the main implications and underlying effects on the performance of the available PPIs to make an informed decision.

Parallel computing researchers have proposed numerous solutions and algorithms to address various aspects of parallelism, including data, task, pipeline, and hybrid parallelism, each with unique characteristics and requirements [60, 4]. Meeting the demands of these applications requires decisions on computing resource allocation, synchronization, communication, and workload distribution strategies. For instance, it is possible to adopt a dynamic or static approach to parallelism, which dramatically changes the application's computational flow and ultimately impacts performance outcomes [18].

Programmability is another crucial characteristic of PPIs [7]. Good programmability is especially relevant at the project decision layer, where consideration of available resources, programmer qualifications, and time-to-deploy is essential. Different abstraction levels, expressiveness, and programming models can impact the productivity of a given PPI. For instance, some interfaces may adopt code annotations that leverage domain-specific knowledge and do not offer much space for customization. In contrast, other PPIs might adopt an explicit approach that allows full customization but may be more cumbersome for simple recurrent tasks. Therefore, another essential decision layer is selecting a PPI that provides a comfortable and adequate abstraction layer between the developer and the low-level concepts of parallelism on limited-resource hardware [27].

Assessing the effectiveness of different parallelism algorithms and PPIs depends on the specific context and goal of the applications. The primary goal of parallelism is to increase computational performance. In this context, data processing applications execute as quickly as possible; real-time applications require low latency; stream processing applications focus on optimizing throughput [8]. In addition to performance improvements, there is a growing awareness of the need to reduce energy consumption and associated costs. Therefore, assessing the impact of parallelism on average and peak power loads is becoming increasingly important, especially for energy grid maintainers [34]. Understanding the effects of parallelism on these aspects is critical for optimizing performance while minimizing energy consumption and costs.

The main **goal** of this Master's thesis is to assess parallelism techniques in the limited-resource hardware environment. This assessment may guide developers when choosing parallelism strategies or interfaces for their applications. Another benefit is to help create new parallel programming interfaces for limited-resource hardware. We start out an assessment with the following question: *what parallelism techniques and interfaces work well for limited-resource hardware?* To that end, we create eight leading research objectives to assess parallel programming in limited-resource hardware. We experiment with 12 parallel applications using six parallel programming interfaces on three limited-resource devices. This methodology allows us to assess multiple parallelism strategies in various circumstances. From there, we asses performance in execution time, energy consumption, and programmability. Therefore, the three leading **scientific contributions** of this Master's thesis are the following:

- Methodology to evaluate the impact of parallelism on limited-resource devices using eight research objectives. This methodology structures parallelism features into programmability, performance in execution time, and energy consumption evaluations. Each research objective presents context, forces, discussion, and threats to the validity of its associated research objective. This methodology can be used by other researchers that aim to structure many tests for parallelism on limited-resource devices.

- A detailed parallelism assessment on multiple limited-resource devices. The discussion is part of our research methodology that goes in-depth about parallelism features and explains the results and behaviors we observe for programmability, performance, and energy consumption.

- Summary of findings in guidelines for efficient parallel programming on limited-resource hardware. We summarize the content of the in-depth discussion into some guidelines or conclusions that can guide the decision-making of parallelism strategies on limited-resource hardware.

The remainder of this document is organized as follows. Chapter 2 provides background information about parallelism, embedded systems, and parallelism interfaces. Chapter 3 discusses related work and the differences between them and this Master's thesis research. Chapter 4 explains the research methodology and experimental setup. Subsequently, Chapter 5 presents an in-depth analysis of this research objectives. The summary of findings and guidelines are presented in Chapter 6. Finally, Chapter 7 discusses final remarks and future works.

2. BACKGROUND

In this Chapter, we present background concepts. We describe details about parallelism and available options in Section 2.1 and explain stream processing in Section 2.2. Then, Section 2.3 explains some embedded systems characteristics and their relation with limited-resource hardware. Finally, we describe SPar in Section 2.4, a high-level DSL we use in our experiments.

2.1 Parallelism

Advances in hardware technology have led to highly transistor-dense chips [83]. The transistor size and spacing could consistently decrease, resulting in increased logical gates in the same silicon area [20, 83]. Clock frequency also increased with each new hardware iteration. However, due to physical constraints such as heat dissipation, current leakage, and high energy consumption, frequency scaling slowed considerably. Manufacturers had to rethink their products' evolution strategy to cope with these challenges. Hardware replication was employed to obtain additional computational power, leading to the rise of multicore processors [4, 36]. Consequently, parallel programming has become essential [76].

Parallelism is a computational method to execute multiple operations simultaneously, leveraging a more significant portion of hardware resources to increase the performance of the computation. Data parallelism, task parallelism, and hybrid are standard methods to classify parallelism. Data parallelism distributes previously known data to parallel computing elements that perform the same operation. Task parallelism applies different operations to different pieces of data simultaneously with a series of data dependencies. Hybrid parallelism is a combination of both, commonly when data parallelism is one of the task parallelism operations.

The most popular algorithm for data parallelism is MapReduce [63]. We demonstrate the execution flow of a vector addition example parallelized using the MapReduce algorithm in Figure 2.1. In this case, the input vector is divided among the parallel threads (1 up to 4); this is the Map phase. Supposing the input vector is larger, the remaining threads can move on to computing other elements. Eventually, the algorithm starts the iterative Reduce phase. After each iteration, the number of computations is reduced in half until only the final output remains. This example is a simple algorithm representation; real PPIs significantly optimize this process.

Despite its importance, parallel programming is a complex endeavor. Its challenges include managing parallel processing units, distributing and balancing data access,

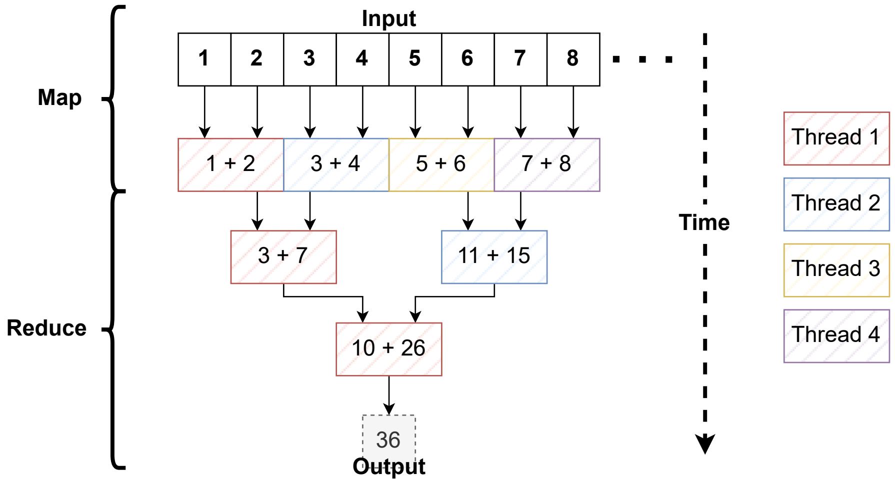


Figure 2.1 – MapReduce algorithm. Vector addition example.

synchronization, control mechanisms, and debugging. Parallel programming complexity is a known challenge in the literature, and, fortunately, many PPIs exist to help ease this burden; however, choosing between may still pose a challenge. Next, we explain the characteristics of some of the existing PPIs in Section 2.1.1.

2.1.1 Parallel Programming Interfaces

PPIs (parallel programming interfaces) help enable parallel processing in their target languages, architectures, and environment. Furthermore, even when they target the same architecture, each PPI has its complexity level, expressiveness, limitations, and performance results. For instance, OpenMP [67] has lower complexity and flexibility than Pthreads [43]. Different runtime parallel systems might perform better in applications with specific computational characteristics since they have different design goals and implementation strategies. Therefore, selecting the appropriate PPI is an important decision that may be specific to each situation.

One strategy widely adopted by PPIs is the structured or parallel patterns approach. They use patterns to codify correct programming practices for specific recurring problems [60]. Examples that adopt this strategy are Intel TBB (Threading Building Blocks) [75], FastFlow [4], Microsoft PPL [62], and some algorithms from the standard C++ STL (Standard Template Library). To the programmer, structured patterns PPIs provide building blocks to instantiate ready-to-use parallel patterns (e.g., Map, Reduce, and Pipeline). These patterns may be nested or mixed to form other more complex parallel patterns. For instance, a user may deploy a pipeline with a Map as one of its internal stages. Most patterns are C++ templates. The user must then refactor the code into their chosen target PPI pattern. Selecting the pattern that best fits the application's computational characteristics is essential. PPIs may also adopt a non-structured approach. Unlike the structured approach that restricts programming to specific and well-defined patterns, with the unstructured approach, the programmer can create code freely if it respects the PPI's syntax and semantics. OpenMP and Pthreads are within this category, but many other examples exist. The flexibility, expressiveness, and complexity may drastically vary between different unstructured PPIs.

DSLs use expert knowledge to provide higher-level abstractions by isolating a specific domain. The language each DSL provides can describe applications expressively and concisely. They can capture specific domain characteristics and use them to develop efficient parallelism while hiding the lower-level architectural and system-dependent details from the developer. The main idea is that the DSL can significantly simplify the process of developing parallel applications by knowing the target hardware and domain characteristics. For the stream processing domain, we can cite Streamit [88] and SPar (detailed later in Section 2.4). From another domain, HIPAcc [61] is a prominent example of an image processing DSL.

CUDA (Compute Unified Device Architecture) [65] is a PPI for parallel computing on NVIDIA GPUs. It allows developers to perform general-purpose computing as well as graphics processing. Programming in CUDA involves creating kernels, which are GPU functions, and host code, which runs on the CPU. The host code transfers data to the kernel and then launches it. Kernels execute on the GPU by potentially hundreds or thousands of threads, which benefits massive parallelism.

The OpenCL (Open Computing Language) [64] PPI is a standard cross-platform heterogeneous parallel programming model. With OpenCL, developers can write a single program that can run on embedded systems and supercomputers. It uses kernels to describe computations that may be compiled and offloaded to different accelerators such as CPU, GPGPU, FPGA, or DSP units. Even though this approach delivers high portability, it also exposes low-level hardware details. Using a generalized and verbose API, the programmer must explicitly define the platform and how work is scheduled onto different devices. Therefore, its language is heavily focused on generalization because its primary design goal is to abstract the underlying hardware to create portable code that can run both on mobile devices and high-end servers. These characteristics make OpenCL an optimal target for higher-level abstractions.

Message-passing programming communicates multiple computing nodes via a network. OpenMPI is an open-source message-passing interface library for distributed computing. It is a standard tool for high-performance distributed programming in C++. The developers create and manage processes and send and receive messages between processes. It is similar to Threads programming because the developer must handle most parallelism algorithmic strategies explicitly. There are other higher-level options for distributed programming, such as Flink [21], a powerful PPI for unified batch and stream processing. Flink may be suitable for distributed dataflow environments to handle data-parallel, event-driven, and pipeline execution flows. Flink may run on a single node and distributed clusters with thousands of processing cores while managed as a local cluster or by resource managers such as Yarn or Kubernetes. Another solution is Storm [86]: a distributed real-time computation system for unbounded streams.

2.2 Stream Processing

The necessity for the stream processing paradigm arises from the constantly growing demand for information-driven or automated digital systems. To meet these demands, industry and academia have made efforts towards developing and deploying stream processing applications in healthcare, transportation, logistics, stock-market, telephony, and many others [8]. Despite the diversity of technologies, these applications share some inherent essential characteristics. They have a constant and potentially infinite data stream flowing through a sequence of independent processing filters. In computer science, applications that consume input to generate an output, like an assembly line, are called pipelines. This paradigm is studied under the stream processing framework.

Stream processing is a paradigm that encompasses the gathering, processing, filtering, and analysis of high-volume, heterogeneous, continuous data streams [8]. Its goal is to reveal valuable information that the data might contain and report back insights, statistics, or suggested courses of action, often with real-time constraints. Failing to meet these constraints may degrade or even destroy the quality of the stream processing application results. Therefore, parallel programming techniques are essential to delivering the highest quality results [88].

Typical stream processing applications can exhibit data, task, and pipeline parallelism, a recurrent subclass of task parallelism. In this case, data parallelism is inside stream processing filters [55]. Data parallelism in stream processing is completely known input data that is divided and processed by parallel agents (i.e., `parallel for` from OpenMP or the `MapReduce` pattern). Task parallelism may be lightweight and short-lived tasks representing work that may be executed asynchronously according to an executor policy. Pipeline parallelism is an assembly line or sequence of independent filters where each filter consumes the data produced by the previous filter as input.

Although there is an abundance of parallelism in stream processing applications, the synchronization and communication costs may eclipse the gains from parallel execution. Choosing the appropriate strategy for each scenario is essential to achieve the promised benefits from parallelism [25]. Regarding stream processing strategies, the main variants are reactive, DataFlow, and stream parallelism. Some scientists must better distinguish between them since they share motivations, goals, main characteristics, and principles. However, the resulting system from them may have slightly different implications.

For reactive stream processing, each actor in the system propagates information and reacts when they receive new data to process. An example may be a web service that must handle click events. This system must handle multiple click sources and react to them immediately. Therefore, parallelism strategies focus on lower latency, data scaling, failure tolerance, and failure recovery.

The DataFlow, also called DataStream, is a flexible model that permits runtime systems to extract parallelism information and act on it based on the available resources, data dependencies, and runtime data. For that, the programmer uses a DAG (Directed Acyclic Graph) programming model to describe the flow of data through the processing filters explicitly. The graph represents data flow as edges and processing filters as vertexes. In this case, the model is highly data-dependent because the data flow within the graph may vary depending on the input. Depending on the data flow, the scheduler could make different scheduling decisions. Examples of PPIs that provide this model are Intel oneAPI [41] with FlowGraph features, OpenStream [73], and ompSS [66] OpenMP extensions, among others.

Stream parallelism describes an explicit execution flow. In this case, computations are a sequence of independent stages or kernels where the previous stage always produces the data for the subsequent stage. The operations inside each stage must be sequential. In contrast to the DataFlow model, the stream parallelism flow is deterministic, significantly simplifying the task scheduler's implementation. Additionally, the system does not need to reason about data globally, which can improve data locality. Stream parallel applications typically prioritize high throughput rates, where examples may be network packet routing, image processing, and data encryption. SPar (later described in Section 2.4) considers only the stream parallelism model.

2.3 Embedded Systems

Embedded systems combine hardware and software designed for a dedicated function [13, 59]. They are in various applications, such as automotive, industrial control, medical devices, smart homes, and mobile devices. Embedded systems possess a different set of requirements and purposes. Some features are computational intensity, memory requirements, energy consumption needs, reliability, and operation lifetime. No hardware configuration fulfills every requirement for all embedded systems. However, one common denominator is the presence of limited-resource hardware.

One of the critical challenges in embedded systems is meeting the performance constraints, often real-time, of the application while also maintaining low instant power drainage. In some systems, computational resources are sacrificed as a trade-off for lower energy consumption or manufacturing price. On the other hand, embedded systems have also become increasingly sophisticated, with many now featuring multicore processors and accelerators such as GPGPUs. Therefore, the embedded systems developer must be proficient in parallel programming, possess good hardware knowledge, write optimized code, and handle software portability, as embedded systems often operate in limited-resource and heterogeneous computational environments.

There has been a growing demand for more powerful and flexible embedded systems that can perform a broader range of functions, specifically in the edge computing research area. The definition of limited-resource hardware may significantly vary. For the scope of this document, we define limited-resource hardware as any single-board computing device equipped with at least one ARM Cortex family processor. The reasoning is twofold: ARM RISC technology is known for its energy efficiency [68]; the Cortex family provides variants for embedded systems focusing on real-time, energy consumption efficiency, micro-controllers, and performance [69]. Next, we discuss possible limited-resource hardware parallel programming optimizations and techniques in Section 2.3.1.

2.3.1 Optimizations and Techniques

This Section examines research that presents optimizations and techniques designed primarily for limited-resource hardware. We consider applications such as cryptography, robotics path-finding, computer vision algorithms, and neural networks. We do not focus on the algorithmic strategies but on the parallelism optimizations or techniques.

Enforcing SLO agreements is a relevant and prevalent topic in the literature. Nornir [82] is a self-adaptive framework to handle the dynamic selection of resources to allocate to the parallel application to enforce performance and energy consumption constraints in shared-memory multicore environments. It goes alongside FastFlow applications and their self-adaptation policies altered by experienced algorithm developers. They monitored throughput and energy consumption with an overhead lower than 1%. They observed that often, depending on the application's stability (data spikes), there is no universally optimal algorithm, and some trade-offs fall onto the programmers' responsibility.

Hsieh et al. [40] presented SURF (Self-aware Unified Runtime Framework) to unify scheduling in the heterogeneous environment from mobile devices. They build on top of OpenMP and OpenCL languages to enable dynamic task mapping to heterogeneous resources based on their system evaluation, runtime measurements, and predictions. The users create SURF tasks that permit dynamic runtime scheduling of computing units (e.g., offload specific tasks to GPGPU or DSP units). Their strategy permitted up to 24% performance improvement over the static alternative on a Qualcomm Snapdragon board (embedded CPU, GPU, and DSP unit). However, there is no abstraction layer since the user must still provide the task source code or binary with the appropriate underlying interface (OpenMP, OpenCL, or DSP code). It focuses mainly on scheduling multiple existing tasks to utilize the available resources better. Their solution is better suited for multiple concurrent tasks.

Aldegheri et al. [3] studied performance improvement methods for computer vision applications on low-energy embedded systems. They combined OpenMP, PThreads, OpenVX, OpenCV, and CUDA programming models to exploit heterogeneous resources available in the system. They propose a template wrapper framework focused on computer vision that encapsulates communication and synchronization features that may be used across their supported programming models. The default Linux scheduler and the GPU scheduling by the OpenVX framework perform the CPU task scheduling. They tested their solution with a robust localization and mapping application for RGB camera sensors. Combining CPU and GPU computations achieves the best energy consumption and performance.

Jubertie et al. [46] evaluated the impact of vectorization and multithreading in Nvidia Jetson boards. They combine software and hardware optimizations to achieve the best trade-off between energy efficiency and performance. They consider multithreading and vectorization implementations on the software and hardware sides, dynamic frequency, and voltage scaling. Considering two Jetson boards, they have measured total energy consumption and speedup in two applications: grayscale conversion and matrix multiplication. However, the GPGPU was not used. They discovered that automatic vectorization could be more reliable for a systematic and coherent speedup. Furthermore, they have found that vectorizing yields superior performance and energy consumption compared to multithreading. However, combining both yields the best results. It is necessary to balance the memory controller and CPU frequency for hardware configurations depending on whether the application is compute- or memory-bound.

Stokke et al. [85] studied the impact of balancing multimedia application workloads between CPU and GPU for energy efficiency. They discovered that saving 5% of energy per frame is possible, while it sometimes degrades energy efficiency. The combination, however, introduced a large amount of extra work with several possible points of failure during the development process. Overall, their findings indicate it is better to select the most suitable processor (CPU or GPU) for each application, which is often only possible to know after development and deployment. In addition, they stated that the best energy efficiency results show when manually stating SIMD instructions. Finally, the optimal results also required finding the minimum operating frequency for the CPU, GPU, and memory controller module that still delivers the specified quality of service.

Lorenzon et al. [53] explored the optimal trade-off between energy consumption and performance considering different thread parallelism levels, number of processes, communication models, and parallel programming interfaces. Energy consumption was estimated based on instruction counters and manufacturer specifications instead of real measurements. Another point is that the dynamic and static power configurations analysis was theoretical, as these are characteristics inherent to hardware manufacturing technology. The authors state that embedded processors are less suitable for ILP (Instruction Level Parallelism) than general-purpose processors but can exploit nearly as much TLP (Thread Level Parallelism). Both TLP and ILP may reduce execution time but increase energy consumption because of extra dynamic power associated with ILP due to extra internal processing units; TLP due to fewer accesses to lower levels of the memory hierarchy. In short, the experiments revealed that an optimal combination does not exist that consistently achieves the best performance with the lowest energy consumption. Selecting the appropriate communication model is often the best alternative.

There are multiple applications targeting embedded system devices. In Artificial Intelligence, Ghadani et al. [2] studied the YOLO CNN Neural Network object detection inference operations and strategies to offload it to embedded GPGPU using an Nvidia TX2 board and Nvidia's TensorRT CUDA SDK. The leading optimization was reducing the memory footprint to the detriment of accuracy. Similarly, Koo et al. [91] parallelized the same application but using OpenCL instead of CUDA achieving a lower performance. Lee et al. [50] provided an efficient SIMD implementation for Convolutional Neural Networks (CNN) targeting limited-resources embedded CPUs. Their optimizations focused on maximizing the use of vectorization instructions using ARM NEON features. Very closely related, another CNN strategy for ARM processors is developed by Zhou et al. [92], where they also used NEON instructions and memory reuse strategies.

Applied to agriculture, the task of retrieving Talreja et al. [87] studied the crop's biophysical parameters. They used an NVIDIA Jetson TK1 board embedded GPGPU to accelerate their application code in R language. On the very particular topic of embedded devices applied to space exploration platforms, Gretok et al. [26] studied OpenMP parallelized applications and their performance in a series of radiation-hardened space-grade processors with shared-memory and multicore characteristics. They used frequency scaling methods, multithreading implementation, and scheduling policy selection to decrease energy consumption and increase performance.

Other critical applications are related to data security. Bahrami et al. [10] provided a parallel implementation of a lightweight data privacy encryption model suiTable for mobile cloud users. They used CUDA to offload it to GPGPU, where they described a 2D array of processes to generate multiple pseudo-random numbers. In this case, they did not care about energy consumption but solely about data security. Another data security study is conducted by Widianto et al. [70] to provide a parallel intrusion detection system based on GPGPU and OpenCL. They concluded that the CPU handles low throughput periods more efficiently while higher throughput spikes should be processed in the GPGPU.

On robotics, Tianji et al. [56] studied an embedded GPGPU solution for autonomic navigation robot field and positioning. They process a stream of data frames captured by a camera positioned in front of the robot using CUDA and an NVIDIA Jetson TX2 board. The main challenge was handling large amounts of data in a limited data storage environment. Related to image processing in robotics, Xie et al. [90] studied a stereo vision algorithm that generates a stream of images to be processed in parallel on low-cost ARM processors. They reduced image sizes by half to the detriment of accuracy and concluded that processing single images at a time instead of batching was beneficial in their case. Essential for robotics or autonomous driving, scene flow applications estimate object geometry and motion based on a set of stereoscopic images. Long Chen et al. [17] studies methodologies to parallelize this type of application on embedded off-the-shelf hardware using OpenCL. They showed the viability of combining data and stream parallelism using a pipeline strategy and achieved near real-time (25 fps) on mobile ARM Mali GPGPU.

In medicine and healthcare, Maheshwari et al. [58] studied an OpenCL-based approach for mapping read operations from genome analysis applications for embedded system devices. This application must typically handle many fragmented reads and present a unique challenge. Their leading optimization was based on dynamic programming filtration techniques to prevent the memory footprint from increasing. The authors further state that it is possible to save energy by moving genomics computing from high-end servers and workstations to embedded systems without losing accuracy and comparable performance.

Boitumelo et. al. [79] parallelized a stereo vision application for UAV (Unmanned Aerial Vehicle). They focus on systems combining embedded ARM processors and GPGPUs, such as NVIDIA Jetson series boards. They combine CUDA-enabled GPGPU parallelism, SIMD instructions using ARM NEON intrinsics, and thread-level CPU parallelism. However, the accuracy results are inconclusive, where one of the benchmarks displays state-of-the-art accuracy and another with 35% error rates. The processing speed and energy consumption were inferior to that of FPGA solutions, partially explained by the FPGA's more efficient data transfer model. However, the development and deployment cycle of the GPGPU is shorter than that of an FPGA. Furthermore, the authors concluded that the SIMD approach uses less energy but sacrifices performance compared to GPGPU. Considering the UAV scenario, where the parallel processing sums up to 1% of the total energy consumption (including UAV rotor), the trade-off of performance per watt of GPGPUs is better than CPU SIMD.

Considering the limited-resource hardware environment, we highlight the main findings and the research that assisted us in inferring them below in no particular order:

- For GPGPU, reducing the memory footprint is of great importance. ([2, 56, 58])
- Certain applications or situations are more suitable to be executed on CPU and others on GPU. ([79, 85, 70])
- Multithreading is important. ([3, 11, 26, 46, 58, 79, 90])
- SIMD (vectorization) is essential for optimal energy consumption and performance. ([46, 50, 79, 85])
- Lower frequencies are generally best for energy consumption, but the appropriate threshold must be established based on application requirements. ([3, 26, 46, 85])
- It is necessary to balance between compute-bound and memory-bound application requirements. ([40, 46])
- Selecting the appropriate communication model is important. ([53])
- Energy can be measured as total consumption in Wh or J, and average power in W. Performance can be measured as execution time, speedup, latency (ms), or throughput (e.g., frames per second). ([3, 10, 46, 85, 70])

2.4 SPar

SPar (an acronym for Stream Parallelism) is a C++ Domain-Specific Language [28]. Figure 2.2 demonstrates a high-level perspective on SPar's usability. In it, the user inserts high-level code annotations in the source code that are automatically interpreted and transformed by SPar to generate the final parallel executable. This methodology aims to provide high-level abstractions for parallel stream processing applications to increase productivity and portability. To better understand SPar, we separate into two concerns: SPar language and SPar compiler (see Figure 2.2). The former is described in Section 2.4.1 and the latter in Section 2.4.2. We finalize by describing ongoing and related SPar research in Section 2.4.3.

2.4.1 SPar Language

The user uses the SPar language to describe essential stream parallelism using source code annotations. This annotation system is the SPar language. The SPar language

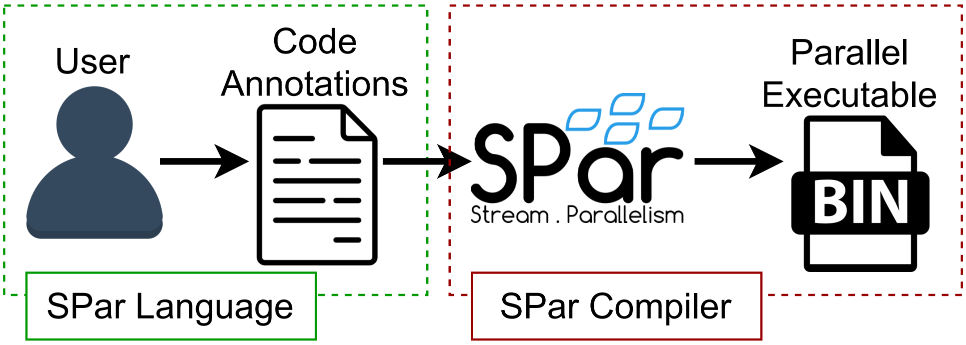


Figure 2.2 – High-Level SPar Usability.

can express standard parallel stream processing application configurations using a basic set of 5 attributes. Data parallelism with SPar (GPGPU or multicore) includes two extra attributes. Default C++ supports the attribute mechanism since C++11 ISO release [42]. These attributes are used within an annotation, delimited by double brackets, and take a list of attributes as input [[attr-list]].

The C++ standard grammar states that annotations can go almost anywhere in the code. This way, SPar language allows describing parallelism while mostly retaining the original sequential code structure. Annotations are an excellent advantage over other lower-level approaches that require sequential code to be re-structured. Regarding the attributes, it is essential to salient that its implementation, in this case, described by the SPar language, will determine its syntax and semantics. Each C++ compiler may support different attributes and implementations for these attributes. Therefore, the need for a compiler to parse the SPar language arises, which will be described in Section 2.4.2. However, every compiler must be able to parse the attributes and place them in the syntax tree, whether it can recognize them. Even if a compiler does not support SPar language, it may still generate regular standard C++ sequential code. Hence, the portability argument.

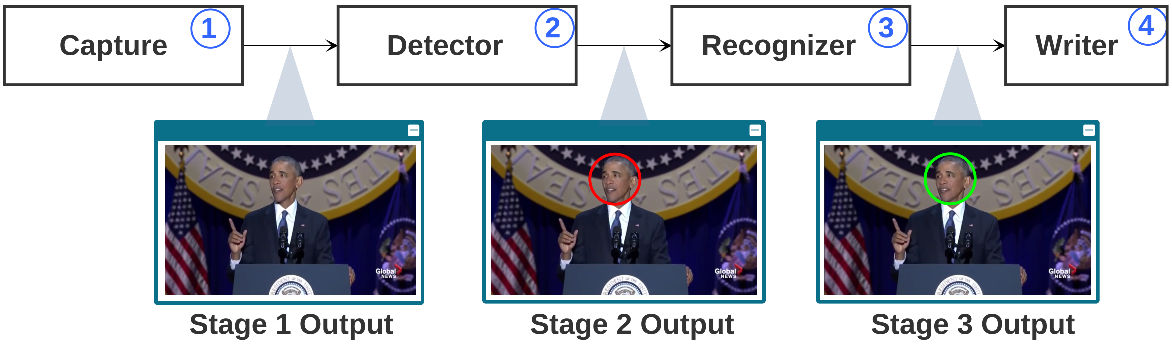
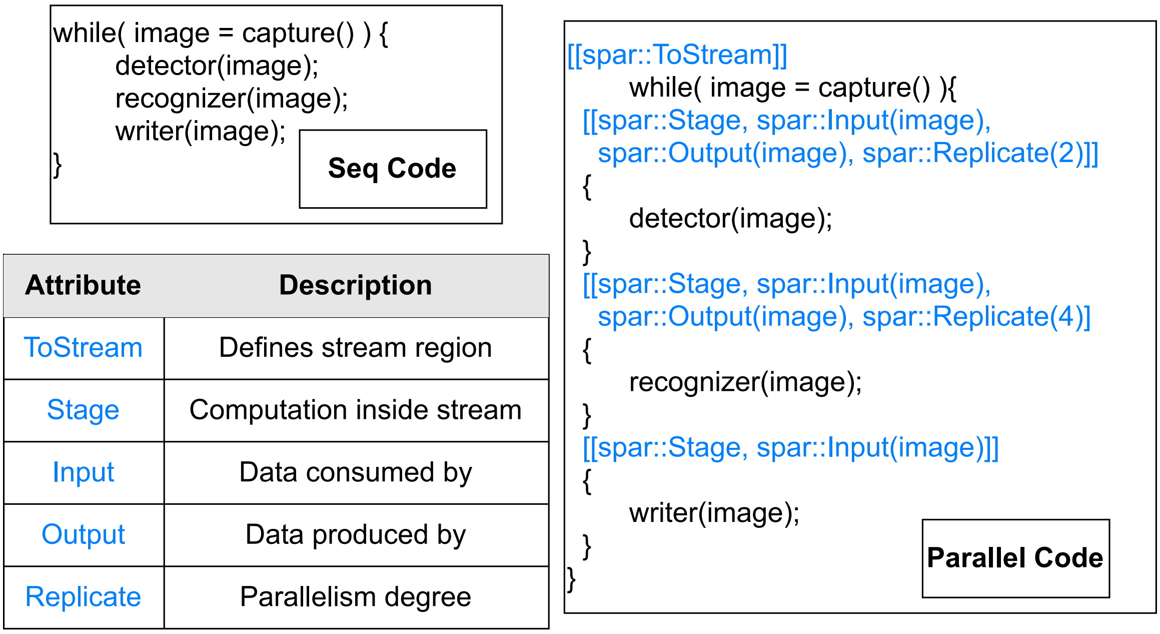


Figure 2.3 – SPar language Face Recognition example. Adapted from [30].

To exemplify the use of SPar language, suppose a SPar user wants to parallelize a face recognition application such as the one presented in Figure 2.3. It receives a video stream as input, detects faces in each frame, recognizes if that face belongs to a person of interest, and then finally writes the output back to the user. The basic sequential code for this example is demonstrated in the top left corner of Figure 2.4. According to SPar good practices guide [27], the user must first examine the sequential code and determine independent stream regions. In this case `capture`, `detector`, `recognizer` and `writer` functions. Secondly, the user must discover which regions may be parallelized or replicated. These are stages that do not share data and are stateless. Stateless stages do not maintain data from previous executions or states. In our example, only the detector and recognizer functions may be parallelized since `capture` and `writer` perform sequential IO operations.



```
while( image = capture() ) {
        detector(image);
        recognizer(image);
        writer(image);
}
```

Attribute	Description
ToStream	Defines stream region
Stage	Computation inside stream
Input	Data consumed by
Output	Data produced by
Replicate	Parallelism degree

```
[[spar::ToStream]]
        while( image = capture() ){
  [[spar::Stage, spar::Input(image),
    spar::Output(image), spar::Replicate(2)]]
  {
        detector(image);
  }
  [[spar::Stage, spar::Input(image),
    spar::Output(image), spar::Replicate(4)]
  {
        recognizer(image);
  }
  [[spar::Stage, spar::Input(image)]]
  {
        writer(image);
  }
}
```

Figure 2.4 – SPar language attributes applied in a Face Recognition code example.

To parallelize the Face Recognition application with SPar language, the user must use a combination of the five attributes briefly described in the Table inside Figure 2.4. To use these attributes, the user inserts as arguments an attribute list inside C++ annotations ([[attr-list]]). This is demonstrated in the right part of Figure 2.4, where the Face Recognition code is parallelized. As a rule, every attribute list must begin with either a `ToStream` or a `Stage` attribute, which SPar described as identifier attributes. Respectively, they describe the stream region and a computational region inside the stream. Every other attribute is auxiliary and is used as a complement of an identifier.

The stream scope is simply defined by the while loop capturing the images. So the `ToStream` is inserted directly above it. The code that generates the data for the rest of the stream (`capture` function) does not need a particular `Stage` and is the only code that can be left outside the scope of a `Stage` inside the `ToStream`. As they are computational stages inside the stream, `detector`, `recognizer`, and `writer` functions are annotated with `Stage` identifier attribute. Also, all of them consume the `image`, so they all add to their complementary list the `Input` attribute with `image` as an argument.

As in Figure 2.3, `detector` outputs its result to `recognizer`, which in turn outputs its computations to the `writer` function. Therefore, the `detector` and `recognizer` must have an `Output` attribute with `image` as an argument. Finally, as the user cleverly discovered, `detector` and `recognizer` may be parallelized, which is achieved by using the `Replicate` attribute with an integer as an argument representing the desired degree of parallelism or how many replicas of that code may exist. Otherwise, every stage that does not have a `Replicate` is considered sequential.

2.4.2 SPar Compiler

The second part of SPar is the compilation process. In short, it receives the annotated code with the SPar language, interprets it, and generates the final parallel code. If there are errors, it reports them back to the user. For these purposes, SPar uses CINCLE compiler infrastructure support [27]. CINCLE (Compiler Infrastructure for new C/C++ Languages Extensions) is a compiler infrastructure for generating new C/C++ embedded DSLs (Domain-Specific Language). It is not a compiler but a support tool that provides basic features and a simple interface to enable AST (Abstract Syntax Tree) transformations, semantic analysis, and source-to-source code generation.

Figure 2.5 depicts the SPar Compiler part of the SPar methodology (see Figure 2.2). Once the user annotates the code with SPar language attributes, the GCC compiler performs a semantic and syntax analysis of default C++ features. Next, CINCLE's scanner forwards the tokens from the original code to the parser. The parser then creates the AST, which is fully accessible and modifiable. This AST contains all of the sequential code and the SPar language attributes. Subsequently, the first SPar module performs the semantic analysis. This step checks the AST for semantic errors in the annotation schema. If present, inconsistencies are reported back to the developer, and the compilation process aborts.

CINCLE's features allow freely removing, shifting, or adding AST nodes, which allows any possible code transformation during compilation time; this is one of the main advantages of CINCLE since GCC and CLANG compilers do not provide direct AST transformations [27]. In a high-level perspective of the transformation phase in Figure 2.5, the SPar compiler removes the annotation nodes and swaps them with nodes containing the appropriate underlying runtime parallel code. However, this step requires extensive attention to the C++ ISO [42], as any error may result in critical failure. On the other hand, regular nodes that describe other application operations remain in their original positions

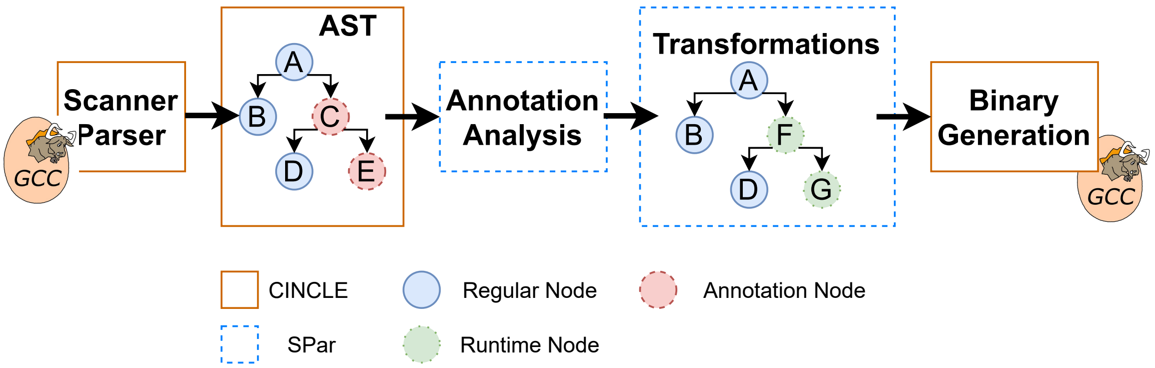


Figure 2.5 – SPar compilation process flow. *Regular orange lines represent CINCLE's modules, and blue dotted lines represent SPar's modules. Additionally, Regular nodes represent the stream operators or filters, Annotation nodes represent SPar's language annotations, and Runtime nodes are the de facto parallel code.*

in the tree. A set of transformation rules guide this entire process which is unique for each underlying runtime parallel system. There are specific rules for FastFlow [28, 55], TBB [38], OpenMP [39], distributed [72], and GPGPU [77]. The final step of the compilation process is the binary code generation from the resulting AST, which uses the system GCC compiler.

SPar also has compilation flags that change the behavior of the stream processing. They are: 1) `spar_ondemand` which changes the scheduling policy from default round-robin to on-demand; 2) `spar_ordered` specifies that a stream region must retain its original generation order (i.e., necessary for video frames that must retain the original order); 3) `spar_blocking` to enable blocking behavior for the FastFlow runtime.

2.4.3 SPar Research

SPar was first developed as a research proof of concept inside the GMAP research group [24] as a doctoral dissertation in 2016 [27]. Over the years, research with SPar has advanced in many directions. Real-world stream processing applications were developed and confirmed to display comparable performance to state-of-the-art solutions [30, 32, 31]. Research with SPar was also conducted to enable support for different underlying runtime and target architectures. For multi-core, it was initially developed with FastFlow [28] and later enabled TBB [38] and OpenMP [39] support. For multicore, efforts were also made to combine data and stream parallelism [51]. Other work enabled SPar support for clusters with DSParLib [52] and GPGPUs with GSParLib [78] runtimes. In the field of self-adaptive parallelism, studies explored an autonomic number of replicas management [89] and service level objectives via annotations [33, 29]. Within the GMAP research group, the work on this Master's thesis is the first to consider limited-resource hardware. It can be a stepping stone for research on parallel computing on limited-resource hardware, particularly expanding the GPGPU parallelism support.

2.5 Examples of Parallel Programming

This Section summarizes programming considerations with all the PPIs we evaluate in this Master's thesis. It helps understand the programmability analysis from Chapter 5. We complement the explanation about SPar from the previous Section 2.4 taking as the basis of example the sequential code for the Face Recognition application. We do not provide an explanation or tutorial of parallelism with each interface. Instead, we describe the fundamentals and provide a visual depiction of the required features. We go into more detail in Chapter 5. In Sections 2.5.1, 2.5.2, 2.5.3, and 2.5.4, we exemplify parallelism using FastFlow, TBB, OpenMP, and Threads.

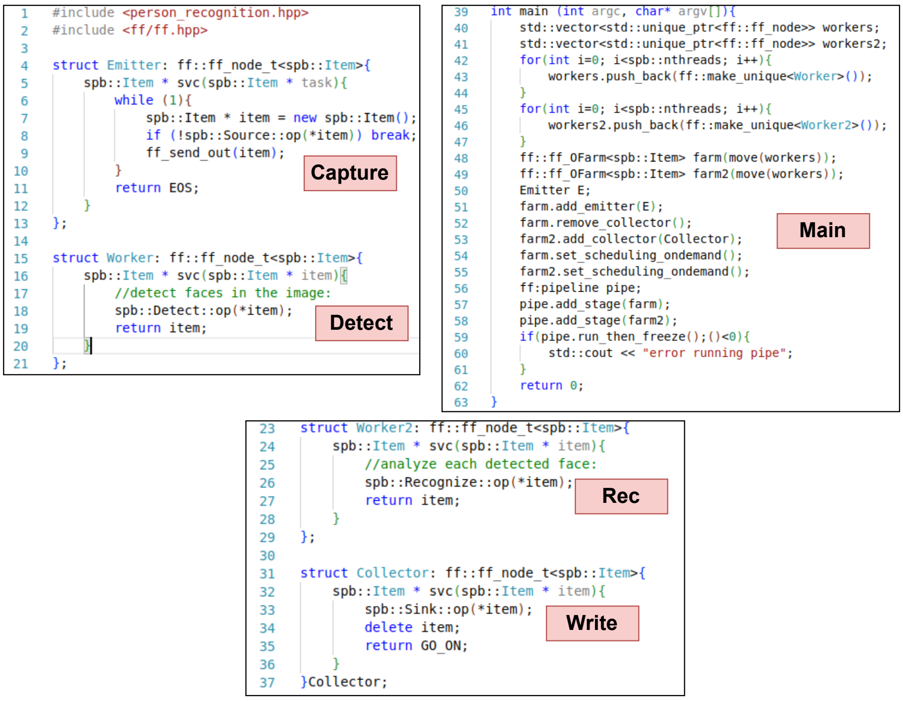
2.5.1 FastFlow

FastFlow [4] is an academic C++ library that adopts a structured approach to parallel programming in multicore, manycore, message-passing, and heterogeneous architectures. FastFlow is built around concurrency patterns such as parallel pipelines, task farms, and data parallelism loops. These patterns allow developers to express parallelism at a high level without worrying about low-level parallelism details. Furthermore, they can combine these patterns to create increasingly complex applications. FastFlow also permits customization of its internal algorithms, such as load balancing.

Figure 2.6 showcases a visual representation of the necessary code to develop a parallel Face Recognizer application using FastFlow. The goal is not to showcase the implementation in detail but to provide a visual representation. Using FastFlow, there are two main steps: 1) organizing the stage code into FastFlow's nodes; 2) creating the parallel patterns on the `Main`, populating them with the processing nodes, configuring the patterns, and then running them.

2.5.2 Threading Building Blocks

TBB (Threading Building Blocks), or oneTBB, is an Intel library for parallel programming on multicore systems [75]. Similarly to FastFlow, it adopts a structured approach to parallel programming that allows users to code at a higher level with pipeline,



```
#include <person_recognition.hpp>
#include <ff/ff.hpp>

struct Emitter: ff::ff_node_t<spb::Item>{
    spb::Item * svc(spb::Item * task){
        while (1){
            spb::Item * item = new spb::Item();
            if (!spb::Source::op(*item)) break;
            ff_send_out(item);
        }
        return EOS;
    }
};

struct Worker: ff::ff_node_t<spb::Item>{
    spb::Item * svc(spb::Item * item){
        //detect faces in the image:
        spb::Detect::op(*item);
        return item;
    }
};
```

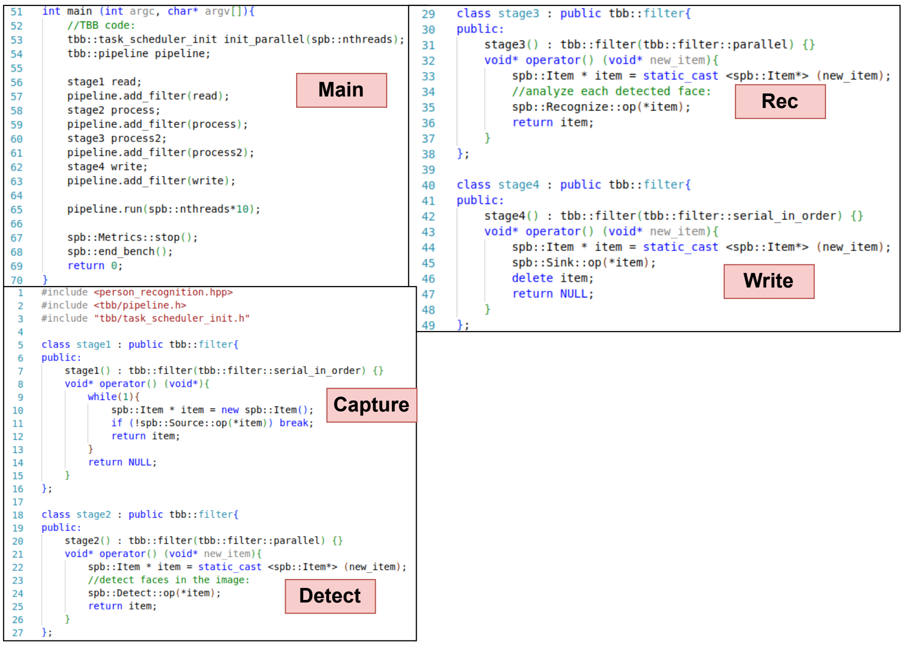
```
int main (int argc, char* argv[]){
    std::vector<std::unique_ptr<ff::ff_node>> workers;
    std::vector<std::unique_ptr<ff::ff_node>> workers2;
    for(int i=0; i<spb::nthreads; i++){
        workers.push_back(ff::make_unique<Worker>());
    }
    for(int i=0; i<spb::nthreads; i++){
        workers2.push_back(ff::make_unique<Worker2>());
    }
    ff::ff_OFarm<spb::Item> farm(move(workers));
    ff::ff_OFarm<spb::Item> farm2(move(workers));
    Emitter E;
    farm.add_emitter(E);
    farm.remove_collector();
    farm2.add_collector(Collector);
    farm.set_scheduling_ondemand();
    farm2.set_scheduling_ondemand();
    ff:pipeline pipe;
    pipe.add_stage(farm);
    pipe.add_stage(farm2);
    if(pipe.run_then_freeze();()<0){
        std::cout << "error running pipe";
    }
    return 0;
}
```

```
struct Worker2: ff::ff_node_t<spb::Item>{
    spb::Item * svc(spb::Item * item){
        //analyze each detected face:
        spb::Recognize::op(*item);
        return item;
    }
};

struct Collector: ff::ff_node_t<spb::Item>{
    spb::Item * svc(spb::Item * item){
        spb::Sink::op(*item);
        delete item;
        return GO_ON;
    }
}Collector;
```

Figure 2.6 – Parallel Face Recognizer visualization of the necessary code with FastFlow.

graph, and data parallelism loops. It adopts a task-based approach to parallel programming that dynamically allocates computational resources and distributes the workload.

Figure 2.6 represents visually the necessary code to develop a parallel Face Recognizer application using TBB. Using TBB is very similar to FastFlow.



```
int main (int argc, char* argv[]){
    //TBB code:
    tbb::task_scheduler_init init_parallel(spb::nthreads);
    tbb::pipeline pipeline;

    stage1 read;
    pipeline.add_filter(read);
    stage2 process;
    pipeline.add_filter(process);
    stage3 process2;
    pipeline.add_filter(process2);
    stage4 write;
    pipeline.add_filter(write);

    pipeline.run(spb::nthreads*10);

    spb::Metrics::stop();
    spb::end_bench();
    return 0;
}
```

```
class stage3 : public tbb::filter{
public:
    stage3() : tbb::filter(tbb::filter::parallel) {}
    void* operator() (void* new_item){
        spb::Item * item = static_cast <spb::Item*> (new_item);
        //analyze each detected face:
        spb::Recognize::op(*item);
        return item;
    }
};

class stage4 : public tbb::filter{
public:
    stage4() : tbb::filter(tbb::filter::serial_in_order) {}
    void* operator() (void* new_item){
        spb::Item * item = static_cast <spb::Item*> (new_item);
        spb::Sink::op(*item);
        delete item;
        return NULL;
    }
};
```

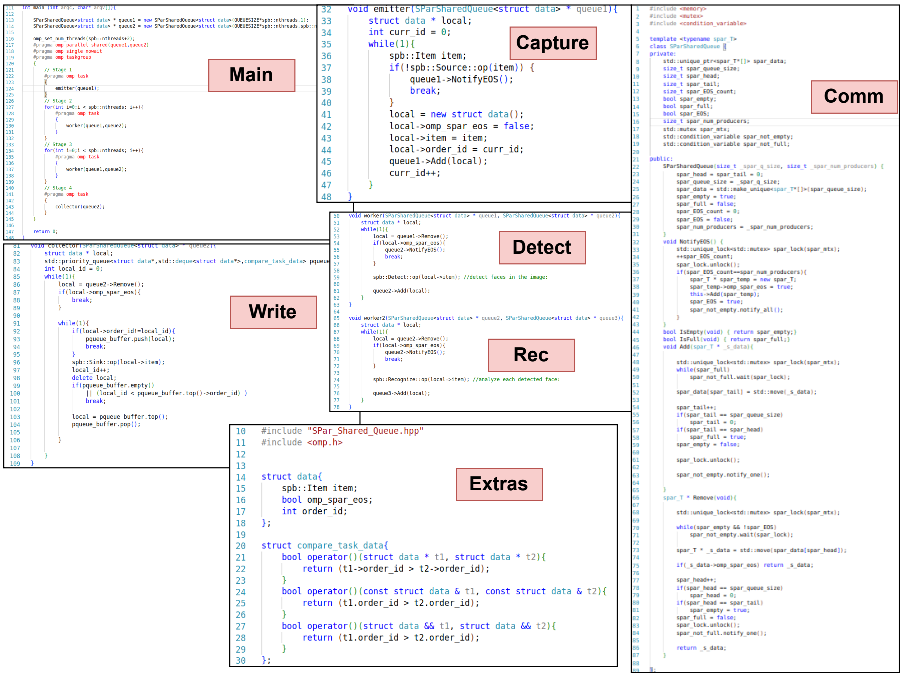
```
#include <person_recognition.hpp>
#include <tbb/pipeline.h>
#include "tbb/task_scheduler_init.h"

class stage1 : public tbb::filter{
public:
    stage1() : tbb::filter(tbb::filter::serial_in_order) {}
    void* operator() (void*){
        while(1){
            spb::Item * item = new spb::Item();
            if (!spb::Source::op(*item)) break;
            return item;
        }
        return NULL;
    }
};

class stage2 : public tbb::filter{
public:
    stage2() : tbb::filter(tbb::filter::parallel) {}
    void* operator() (void* new_item){
        spb::Item * item = static_cast <spb::Item*> (new_item);
        //detect faces in the image:
        spb::Detect::op(*item);
        return item;
    }
};
```

Figure 2.7 – Parallel Face Recognizer visualization of the necessary code with TBB.

2.5.3 OpenMP

OpenMP uses a set of directives, runtime libraries, that enable the programmer to specify how a parallel program is divided among multiple processing elements [67]. Therefore, it is well suited for data processing applications, although it has limited support for task and stream parallelism [73, 39]. OpenMP parallelism is achieved using pragma notations on the code that are then interpreted by a compiler that implements the OpenMP standard.

Figure 2.6 showcases a visual representation of the necessary code to develop a parallel Face Recognizer application using OpenMP. For stream processing applications like Face Recognizer, the user has to define the communication protocol [37] and data re-ordering algorithm [31]. Other than that, OpenMP requires embedding the communication protocol into the default application logic and spawning Sections of asynchronous tasks on the main function. The extras represent support codes for re-ordering and communication.



```
int main (int argc, char* argv[]){

    SParSharedQueue<struct data> * queue1 = new SParSharedQueue<struct data>(QUEUESIZE*spb::nthreads,1);
    SParSharedQueue<struct data> * queue2 = new SParSharedQueue<struct data>(QUEUESIZE*spb::nthreads,spb::n

    omp_set_num_threads(spb::nthreads+2);
    #pragma omp parallel shared(queue1,queue2)
    #pragma omp single nowait
    #pragma omp taskgroup
    {
        // Stage 1
        #pragma omp task
        {
            emitter(queue1);
        }
        // Stage 2
        for(int i=0;i < spb::nthreads; i++){
            #pragma omp task
            {
                worker(queue1,queue2);
            }
        }
        // Stage 3
        for(int i=0;i < spb::nthreads; i++){
            #pragma omp task
            {
                worker(queue1,queue2);
            }
        }
        // Stage 4
        #pragma omp task
        {
            collector(queue2);
        }
    }

    return 0;
}
```

```
void emitter(SParSharedQueue<struct data> * queue1){
    struct data * local;
    int curr_id = 0;
    while(1){
        spb::Item item;
        if(!spb::Source::op(item)) {
            queue1->NotifyEOS();
            break;
        }
        local = new struct data();
        local->omp_spar_eos = false;
        local->item = item;
        local->order_id = curr_id;
        queue1->Add(local);
        curr_id++;
    }
}
```

```
void worker(SParSharedQueue<struct data> * queue1, SParSharedQueue<struct data> * queue2){
    struct data * local;
    while(1){
        local = queue1->Remove();
        if(local->omp_spar_eos){
            queue2->NotifyEOS();
            break;
        }

        spb::Detect::op(local->item); //detect faces in the image:

        queue2->Add(local);
    }
}

void worker2(SParSharedQueue<struct data> * queue2, SParSharedQueue<struct data> * queue3){
    struct data * local;
    while(1){
        local = queue2->Remove();
        if(local->omp_spar_eos){
            queue2->NotifyEOS();
            break;
        }

        spb::Recognize::op(local->item); //analyze each detected face:

        queue3->Add(local);
    }
}
```

```
void collector(SParSharedQueue<struct data> * queue2){
    struct data * local;
    std::priority_queue<struct data*,std::deque<struct data*>,compare_task_data> pqueue
    int local_id = 0;
    while(1){
        local = queue2->Remove();
        if(local->omp_spar_eos){
            break;
        }

        while(1){
            if(local->order_id!=local_id){
                pqueue_buffer.push(local);
                break;
            }
            spb::Sink::op(local->item);
            local_id++;
            delete local;
            if(pqueue_buffer.empty()
                || (local_id < pqueue_buffer.top()->order_id) )
                break;

            local = pqueue_buffer.top();
            pqueue_buffer.pop();

        }

    }
}
```

```
#include "SPar_Shared_Queue.hpp"
#include <omp.h>


struct data{
    spb::Item item;
    bool omp_spar_eos;
    int order_id;
};

struct compare_task_data{
    bool operator()(struct data * t1, struct data * t2){
        return (t1->order_id > t2->order_id);
    }
    bool operator()(const struct data & t1, const struct data & t2){
        return (t1.order_id > t2.order_id);
    }
    bool operator()(struct data && t1, struct data && t2){
        return (t1.order_id > t2.order_id);
    }
};
```

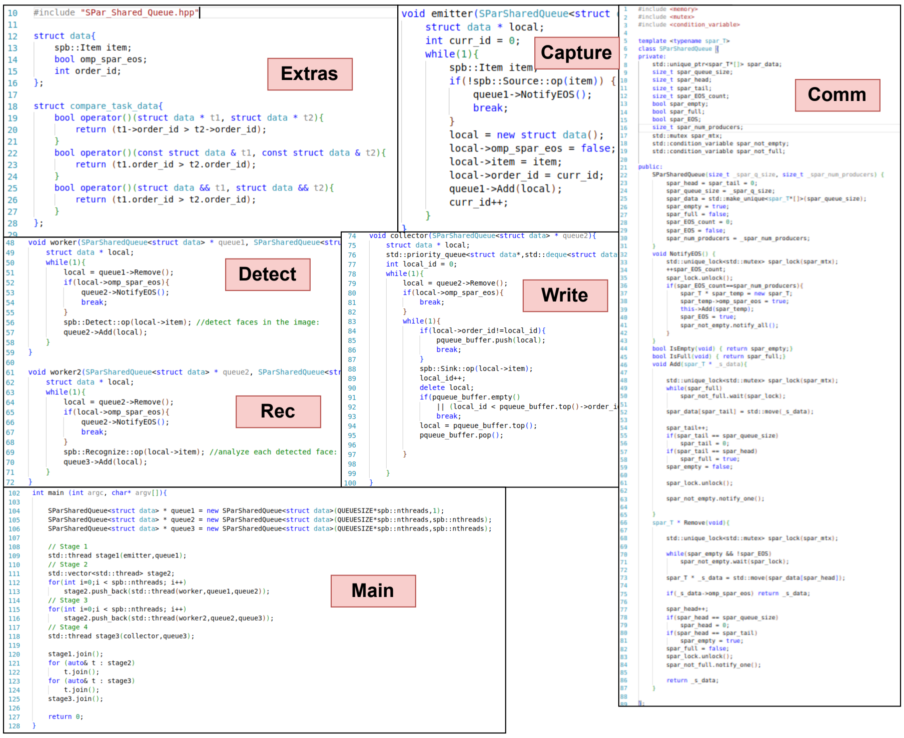
```
#include <memory>
#include <mutex>
#include <condition_variable>

template <typename spar_T>
class SParSharedQueue {
private:
    std::unique_ptr<spar_T*[]> spar_data;
    size_t spar_queue_size;
    size_t spar_head;
    size_t spar_tail;
    size_t spar_EOS_count;
    bool spar_empty;
    bool spar_full;
    bool spar_EOS;
    size_t spar_num_producers;
    std::mutex spar_mtx;
    std::condition_variable spar_not_empty;
    std::condition_variable spar_not_full;

public:
    SParSharedQueue(size_t _spar_q_size, size_t _spar_num_producers) {
        spar_head = spar_tail = 0;
        spar_queue_size = _spar_q_size;
        spar_data = std::make_unique<spar_T*[]>(spar_queue_size);
        spar_empty = true;
        spar_full = false;
        spar_EOS_count = 0;
        spar_EOS = false;
        spar_num_producers = _spar_num_producers;
    }
    void NotifyEOS() {
        std::unique_lock<std::mutex> spar_lock(spar_mtx);
        ++spar_EOS_count;
        spar_lock.unlock();
        if(spar_EOS_count==spar_num_producers){
            spar_T * spar_temp = new spar_T;
            spar_temp->omp_spar_eos = true;
            this->Add(spar_temp);
            spar_EOS = true;
            spar_not_empty.notify_all();
        }
    }
    bool IsEmpty(void) { return spar_empty;}
    bool IsFull(void) { return spar_full;}
    void Add(spar_T * _s_data){

        std::unique_lock<std::mutex> spar_lock(spar_mtx);
        while(spar_full)
            spar_not_full.wait(spar_lock);

        spar_data[spar_tail] = std::move(_s_data);

        spar_tail++;
        if(spar_tail == spar_queue_size)
            spar_tail = 0;
        if(spar_tail == spar_head)
            spar_full = true;
        spar_empty = false;

        spar_lock.unlock();

        spar_not_empty.notify_one();

    }
    spar_T * Remove(void){

        std::unique_lock<std::mutex> spar_lock(spar_mtx);

        while(spar_empty && !spar_EOS)
            spar_not_empty.wait(spar_lock);

        spar_T * _s_data = std::move(spar_data[spar_head]);

        if(_s_data->omp_spar_eos) return _s_data;

        spar_head++;
        if(spar_head == spar_queue_size)
            spar_head = 0;
        if(spar_head == spar_tail)
            spar_empty = true;
        spar_full = false;
        spar_lock.unlock();
        spar_not_full.notify_one();

        return _s_data;
    }

};
```

Figure 2.8 – Parallel Face Recognizer visualization of the necessary code with OpenMP.

2.5.4 Threads

Thread [43] programming is present in most modern languages and is used to develop multithreaded applications. Threads allow developers to write code that can perform multiple tasks simultaneously, leveraging multicore computers. However, developers have to explicitly define algorithms for parallelism, directly controlling resource creation, communication, load balancing, and synchronization. For that, programmers are equipped with tools for creating, managing, and synchronizing threads, including thread pools, locks, semaphores, and barriers. The vast customization space permits highly-specialized code; however, it permits every pitfall of parallel programming, such as race conditions, deadlocks, and unoptimized implementation strategies.

Figure 2.6 showcases a visual representation of the necessary code to develop a parallel Face Recognizer application using Threads. Using Threads, developers have to specify all aspects of parallelism. In this case, the most costly implementations are re-ordering and communication protocols, which are the same as OpenMP. Then, the user must also define the functions each thread will execute. Finally, it spawns and joins a static number of threads on the main function.



```
#include "SPar_Shared_Queue.hpp"

struct data{
    spb::Item item;
    bool omp_spar_eos;
    int order_id;
};

struct compare_task_data{
    bool operator()(struct data * t1, struct data * t2){
        return (t1->order_id > t2->order_id);
    }
    bool operator()(const struct data & t1, const struct data & t2){
        return (t1.order_id > t2.order_id);
    }
    bool operator()(struct data && t1, struct data && t2){
        return (t1.order_id > t2.order_id);
    }
};
```

```
void emitter(SParSharedQueue<struct
    struct data * local;
    int curr_id = 0;
    while(1){
        spb::Item item;
        if(!spb::Source::op(item)) {
            queue1->NotifyEOS();
            break;
        }
        local = new struct data();
        local->omp_spar_eos = false;
        local->item = item;
        local->order_id = curr_id;
        queue1->Add(local);
        curr_id++;
    }
}
```

```
#include <memory>
#include <mutex>
#include <condition_variable>

template <typename spar_T>
class SParSharedQueue {
private:
    std::unique_ptr<spar_T*[]> spar_data;
    size_t spar_queue_size;
    size_t spar_head;
    size_t spar_tail;
    size_t spar_EOS_count;
    bool spar_empty;
    bool spar_full;
    bool spar_EOS;
    size_t spar_num_producers;
    std::mutex spar_mtx;
    std::condition_variable spar_not_empty;
    std::condition_variable spar_not_full;

public:
    SParSharedQueue(size_t _spar_q_size, size_t _spar_num_producers) {
        spar_head = spar_tail = 0;
        spar_queue_size = _spar_q_size;
        spar_data = std::make_unique<spar_T*[]>(spar_queue_size);
        spar_empty = true;
        spar_full = false;
        spar_EOS_count = 0;
        spar_EOS = false;
        spar_num_producers = _spar_num_producers;
    }
    void NotifyEOS() {
        std::unique_lock<std::mutex> spar_lock(spar_mtx);
        ++spar_EOS_count;
        spar_lock.unlock();
        if(spar_EOS_count==spar_num_producers){
            spar_T * spar_temp = new spar_T;
            spar_temp->omp_spar_eos = true;
            this->Add(spar_temp);
            spar_EOS = true;
            spar_not_empty.notify_all();
        }
    }
    bool IsEmpty(void) { return spar_empty;}
    bool IsFull(void) { return spar_full;}
    void Add(spar_T * _s_data){

        std::unique_lock<std::mutex> spar_lock(spar_mtx);
        while(spar_full)
            spar_not_full.wait(spar_lock);

        spar_data[spar_tail] = std::move(_s_data);

        spar_tail++;
        if(spar_tail == spar_queue_size)
            spar_tail = 0;
        if(spar_tail == spar_head)
            spar_full = true;
        spar_empty = false;

        spar_lock.unlock();

        spar_not_empty.notify_one();

    }
    spar_T * Remove(void){

        std::unique_lock<std::mutex> spar_lock(spar_mtx);

        while(spar_empty && !spar_EOS)
            spar_not_empty.wait(spar_lock);

        spar_T * _s_data = std::move(spar_data[spar_head]);

        if(_s_data->omp_spar_eos) return _s_data;

        spar_head++;
        if(spar_head == spar_queue_size)
            spar_head = 0;
        if(spar_head == spar_tail)
            spar_empty = true;
        spar_full = false;
        spar_lock.unlock();
        spar_not_full.notify_one();

        return _s_data;
    }

};
```

```
void worker(SParSharedQueue<struct data> * queue1, SParSharedQueue<stru
    struct data * local;
    while(1){
        local = queue1->Remove();
        if(local->omp_spar_eos){
            queue2->NotifyEOS();
            break;
        }
        spb::Detect::op(local->item); //detect faces in the image:
        queue2->Add(local);
    }
}

void worker2(SParSharedQueue<struct data> * queue2, SParSharedQueue<str
    struct data * local;
    while(1){
        local = queue2->Remove();
        if(local->omp_spar_eos){
            queue2->NotifyEOS();
            break;
        }
        spb::Recognize::op(local->item); //analyze each detected face:
        queue3->Add(local);
    }
}
```

```
void collector(SParSharedQueue<struct data> * queue2){
    struct data * local;
    std::priority_queue<struct data*,std::deque<struct data
    int local_id = 0;
    while(1){
        local = queue2->Remove();
        if(local->omp_spar_eos){
            break;
        }
        while(1){
            if(local->order_id!=local_id){
                pqueue_buffer.push(local);
                break;
            }
            spb::Sink::op(local->item);
            local_id++;
            delete local;
            if(pqueue_buffer.empty()
                || (local_id < pqueue_buffer.top()->order_i
                break;
            local = pqueue_buffer.top();
            pqueue_buffer.pop();

        }

    }
}
```

```
int main (int argc, char* argv[]){

    SParSharedQueue<struct data> * queue1 = new SParSharedQueue<struct data>(QUEUESIZE*spb::nthreads,1);
    SParSharedQueue<struct data> * queue2 = new SParSharedQueue<struct data>(QUEUESIZE*spb::nthreads,spb::nthreads);
    SParSharedQueue<struct data> * queue3 = new SParSharedQueue<struct data>(QUEUESIZE*spb::nthreads,spb::nthreads);

    // Stage 1
    std::thread stage1(emitter,queue1);
    // Stage 2
    std::vector<std::thread> stage2;
    for(int i=0;i < spb::nthreads; i++)
        stage2.push_back(std::thread(worker,queue1,queue2));
    // Stage 3
    for(int i=0;i < spb::nthreads; i++)
        stage2.push_back(std::thread(worker2,queue2,queue3));
    // Stage 4
    std::thread stage3(collector,queue3);

    stage1.join();
    for (auto& t : stage2)
        t.join();
    for (auto& t : stage3)
        t.join();
    stage3.join();

    return 0;
}
```

Figure 2.9 – Parallel Face Recognizer visualization of the necessary code with Threads.

3. RELATED WORK

This Chapter discusses related work to this Master's thesis research. We elect the Scopus scientific literature database. Figure 3.1 showcases our final search string. It searches titles, abstracts, and keywords for parallelism evaluations on limited-resource hardware considering energy, performance, or programmability and limits to benchmarking evaluations on the computer science sub-area. We also consider research from 2016 onward. To exclude or include articles, we use a set of inclusion criteria (IC) and exclusion criteria (EC) summarized in Table 3.1. A document must pass all the CI to be evaluated in detail. However, if even one of the CE is applicable, the document in question is disregarded. Out of 51 documents found within our parameters, using our IC and EC, we include only seven documents that passed our screening. We include research not listed in our initial literature review search to complement and enhance our related work.

```
TITLE-ABS-KEY ( ( "Parallel*" )
AND ( "Cortex" OR "ARM" )
AND ( "energy" OR "performance" )
AND ( "assessment" OR "benchmark" ) )
AND PUBYEAR > 2015 AND PUBYEAR < 2024
AND ( LIMIT-TO ( SUBJAREA , "COMP" ) )
AND ( LIMIT-TO ( LANGUAGE , "English" ) )
AND ( LIMIT-TO ( EXACTKEYWORD , "Benchmarking" ) )
```

Figure 3.1 – Search string on Scopus scientific database.

Table 3.1 – Criteria for selecting documents in the Literature Review process.

ID	**Inclusion Criteria**
IC-1	Considers data or stream applications
IC-2	Considers parallel computing
IC-3	Experiments with ARM-based hardware
IC-4	Measures performance or energy
ID	**Exclusion Criteria**
EC-1	Simulated hardware
EC-2	Duplicated research
EC-3	Brief summary
EC-4	Unable to access research

Next, we discuss related research in Section 3.1 and then compare them with this Master's thesis research in Section 3.2.

3.1 Related Research

Clemons et al. [19] introduced MEVBench, a benchmark suite designed to evaluate mobile embedded vision architectures through various workloads. The authors evaluate the performance of MEVBench on various platforms and processors to gain insights into possible future mobile-embedded vision architectures. The authors highlight the necessity and performance improvements of data parallelism computations but do not consider stream parallelism algorithms. They also assess the cache miss rate and the importance of optimal data access patterns. They do not assess power dissipation or energy consumption but measure performance using clock cycles.

Magnussem et al. [57] present OSCAR, a compiler for automatic parallelization. They describe their compiler strategy, which generates OpenMP parallel code. They also perform experiments on a series of traditional HPC and limited-resource hardware. Using four cores on the limited-resource hardware, they achieved speedups of 2.87 for the CG program of the NAS parallel benchmarks and 2.64 and 1.86 for other data processing benchmarks. Ultimately, they observe that their compiler can generate efficient OpenMP code for limited-resource hardware.

Canizzaro et al. [16] evaluates the performance and power dissipation of RISC-V architectures compared to ARM Cortex A9 and A53. The RISC-V architecture demonstrates competitive single-core performance, high efficiencies in CoreMark, and average multi-core performance results when evaluated under embedded system-focused workloads, including SpaceBench and GKSuite. The study also observes that the RISC-V architecture's dynamic power and energy consumption show mixed results, with high dynamic energy consumption during many SpaceBench algorithms and good power dissipation results during CoreMark application benchmarking tests.

Amiri et al. [5] present two scheduling algorithms for simultaneous multiprocessing using an ARM Cortex A53 with an FPGA setup. The algorithms are evaluated using four benchmarks with different exceptional features. They use TBB to enable parallelism on the A53 side and offload some of the computation to the FPGA. To balance computation between CPU and FPGA, they proposed two algorithms. However, their main limitation is that they must compare their solution with the sequential or parallel CPU versions.

Belloch et al. [14] study the feasibility of combining multiple processing units of system-on-chip hardware to accelerate image-based applications. Combining ARM Cortex A53 and R5 with an FPGA in imaged-based applications, they achieve an acceleration factor of 12x in the number of frames per second. The results show that the ARM Cortex-A53 performs best regarding MFlops per second for matrix multiplication computation. At the same time, the GPU is better suited for graphic processing tasks, respectively. The study also evaluates the performance of A53 using the OpenBLAS library and optimized parallel code for matrix multiplication, achieving up to 12 Gflops.

Chronaki et al. [18] evaluated Parsec benchmark applications (data and stream processing) on ARM with BIG.little architectural features. Their main goal is to evaluate the portability of the traditional HPC applications to the BIG.little architecture features. To that end, they evaluate OS scheduling (10% improvement) and custom runtime scheduling (23% improvement). They evaluate speedup, energy consumption, and power dissipation on an Odroid-XU4 device containing Cortex A15 and Cortex A7; however, they do not explain their methodology to collect energy measurements. They conclude that current implementations of parallel applications using Pthreads are not ready to fully leverage BIG.little architectural features, except for applications with highly sophisticated parallelization strategies. The study also shows better solutions than a highly sophisticated asymmetry-aware OS scheduler for scheduling parallel applications. On the other hand, a task-based solution, even if it is asymmetry-unaware, is the most appropriate as it allows dynamic load balancing and eliminates thread migration costs.

Simula et al. [84] presented a study on the computational cost and power efficiency of neural simulations on a series of hardware, including a limited-resource cortex A53 cluster. Parallelism is developed with OpenMPI support. However, their energy measurements are estimated. They report that the DPSNN simulator running on ARM A53 requires about 3× less energy consumption than Intel but is about 5× slower. The main limiting factor to parallelism scalability was the interconnect speed.

Görtz et al. [35] study the viability of using a cluster of 40 Cortex A53 devices to assess its computational and energy efficiency compared to an Intel cluster. Experimenting with NPB, they report up to 70% less energy than an Intel-based reference server system, which leads to an increase in efficiency of up to 425%. They also highlight that keeping an ideal operating temperature is a challenge. Khasanov et al. [48] present an extension to the KPN model (calculates Mandelbrot set) and a simple execution strategy for data parallelism. They evaluate their approach on an ARM big.LITTLE architecture. However, they do not provide many details about the parallelism interfaces used.

Junior et al. [47] evaluated energy consumption and performance of the NPB benchmark on two clusters, one containing 8 Cortex A7 and the other containing 4 Cortex A15. Similar to our experiments, they executed NPB's five kernels and two pseudo-applications with input class B and used OpenMP for parallelism. Their results indicated that the Cortex A15 mounted on a Jetson board with 2 GB of ram outperformed cortex A7 in energy- and time-to-solution. In contrast, we focus on analyzing the impact of parallelism in the NPB benchmark and include different parallel APIs. In addition, we also use two different more up-to-date hardware configurations that are standalone shared-memory processors that are not bound to a desktop-controlled cluster.

Schmid et al. [81] studied the feasibility of task model parallelism on a system with 8 Cortex A7 processors and 2 GB of RAM. They experimented with TBB and their lightweight task parallelism model emphasizing minimizing memory space consumption. They executed FFT (Fast Fourier Transform) and matrix multiplication applications. However, their experiments could have provided a more precise performance comparison between parallel versions.

Rauber et al. [74] evaluated performance and energy consumption in two benchmark suites that combine applications from multiple domains. Their analysis combines frequency scaling and thread parallelism, emphasizing energy consumption. They tested on an ARM A15 and A7 big.LITTLE setup with 2 GB of RAM. They only executed some of the applications on the ARM setup since they used it as a complementary study. They observed lower energy costs for multi-thread use but higher overall execution time and energy consumption. In contrast to our work, they did not assess parallelism features.

Jubertie et al. [46] evaluated vectorization and multithreading in an ARM Cortex A15 board. They experimented with two applications: grayscale conversion and matrix multiplication. They discovered that combining vectorization and multithreading yields the best results in energy consumption and execution time. Similarly, Lee et al. [50], and Zhou et al. [92] experimented with Convolutional Neural Networks (CNN). They focused on generating vectorized ARM NEON instructions and minimizing memory usage. They did not experiment with multiple parallelism interfaces.

Xie et al. [90] studied a stereo vision algorithm that generates a stream of images to be processed in parallel on low-cost ARM processors. They reduced image sizes by half to the detriment of accuracy and concluded that processing single images at a time instead of batching was beneficial in their case. In medicine and healthcare, Maheshwari et al. [58] studied an OpenCL-based approach for mapping read operations from genome analysis applications for embedded system devices. These applications must typically handle many fragmented reads and present a unique challenge. Their leading optimization was based on dynamic programming filtration techniques to prevent the memory footprint from increasing. The authors further state that it is possible to save energy by moving genomics computing from high-end servers and workstations to embedded systems without losing accuracy and comparable performance.

3.2 Comparison with this work

Table 3.2 summarizes the differences between our research and previously discussed related work. The `Data` and `stream` fields indicate each research experiment's data and stream processing applications. For `stream`, FE is Ferret, BZ2 is Bzip2, LD is Lane Detection, and FR is Face Recognizer. `Parallelism` indicates the parallel interfaces evaluated and `Architecture` specifies hardware used in the experiments and **Accelerator** highlights if that research considers GPGPU or FPGA accelerators. `Metrics` summarizes the measurements taken into consideration, where time performance is related to the total application execution time, energy is the total consumption in Wh, and programmability assesses estimated development complexity.

Table 3.2 – Comparing related work with this Master's thesis research.

Research	Data	Stream	Parallelism	Architecture	Accelerator	Metrics
Ours	**NPB**	**FE, BZ2, LD, FR**	**OpenMP, Threads, Intel TBB, FastFlow, SPar, CUDA, and OpenCL**	**Cortex A57 Cortex A72 Cortex A53 Cortex A73**	**Maxwell Tegra Mali G52**	**Time & Energy & Programmability**
[47]	NPB	None	OpenMP	Cortex A7 A15 Clusters	None	Time & Energy
[81]	FFT & Matrix Mult	None	TBB	Cortex A7	None	Time
[74]	None	Partial Parsec	Pthreads	Cortex A15 Corte A7	None	Time & Energy
[46]	Grayscale Matrix Mult	None	-	Cortex A15	None	Time & Energy
[50]	CNN	None	Neon & OpenMP	Cortex A53	None	Time & Energy
[92]	CNN	None	Neon	Cortex A17	None	Time
[90]	Stereo Vision	None	OpenMP	Odroid XU4	None	Time
[58]	Genomic	None	OpenCL	Cortex A73 & A53	None	Time & Energy
[19]	MEVBench	None	-	Cortex A9 Intel Atom	None	Time
[57]	Partial NPB & More	None	OpenMP	NVIDIA Carmel ARMv8	None	Time
[16]	SpaceBench GKSuiteCoreMark	None	OpenMP	Cortex A53 Cortex A9	None	Time & Energy
[5]	AES, HotSpot, Nbody, GEMM	None	TBB	Cortex A53	FPGA	Time & Energy
[14]	Image-processing Matrix Mult	None	OpenMP OpenCV (TBB, OpenMP)	Cortex A53 Cortex R5	FPGA GPU	Time
[18]	Parsec	Parsec	OpenMP Pthreads	Cortex A15 Cortex A7	None	Time & Energy
[84]	Neural Network	None	OpenMPI	Cortex A53 Cluster	None	Time
[35]	Partial NPB	None	-	Cortex A53 Cluster	None	Time & Energy
[48]	Mandelbrot	None	-	Cortex A15 Cortex A7	None	Time

In Table 3.2, our research and [18] are the only ones to experiment with data and stream parallelism applications on limited-resource hardware. Few related work study stream processing applications, and the ones that do only use two Parsec applications. On the other hand, our research uses four applications, which we describe in greater detail in Section 4.3. Regarding parallelism, we are the only researchers using more than two parallelism interfaces. This study experiments with OpenMP, Threads, Intel TBB, FastFlow, SPar, CUDA, and OpenCL. Additionally, OpenMP is the most common interface other researchers use. Our research is the only test with multiple hardware configurations on the target architecture side, as well as two GPGPU accelerators. For metrics, we include data and stream processing typical metrics such as execution time, energy, and latency. Not highlighted in the Table, our work is the only one that provides detailed Research Objectives to assess parallelism on limited-resource hardware and concisely summarizes them in a summary of findings.

4. METHODOLOGY

This Chapter describes this Master's Thesis research methodology. First, we outline our Research Objectives in Section 4.1. Then we describe the hardware and software environment we use for our analysis in Section 4.2. Section 4.3 describes every application we use to experiment with our research objectives, while Section 4.4 describes and explains the metrics we use in our assessment. We also explain the parallel application interfaces we use in Section 4.5 and describe the experimental execution methodology and graph patterns in Section 4.6.

4.1 Research Objectives

The primary goal of this Master's thesis is to assess parallelism techniques in limited-resource hardware. We start with the leading research question: *What parallelism algorithms and interfaces are best for limited-resource hardware?* Then, we formulate **Research Objectives** (ROs) to direct our experimental analysis and draw relevant conclusions. Following this paragraph, we list each RO we examine in greater detail in the next Chapter 5. We have eight original leading ROs; some ROs have derived ROs, which we created as the research progressed to separate concerns about the leading RO better.

- RO1: Evaluating Programming features for the parallel programming interfaces
- RO2: Evaluating Programmability metrics for the parallel programming interfaces
- RO3: Evaluating Performance metrics for data processing applications
- RO3.1: Analysis of MapReduce with no data dependencies
- RO3.2: Evaluating Map with memory contention
- RO3.3: Evaluating data locality on Maps
- RO3.4: Evaluating Map with data dependency
- RO3.5: Analyzing memory bandwidth as a parallelism bottleneck
- RO3.6: BIG.little architecture performance features
- RO3.7: Analysis of the impact of hardware temperature on performance
- RO4: Evaluating consumption metrics for data processing applications
- RO5: Evaluating performance metrics of stream processing applications

- RO5.1: Assessing the effect of blocking vs. non-Blocking computation on performance metrics
- RO5.2: Evaluating computational resource allocation strategies in parallelism techniques
- RO6: Evaluating consumption metrics for stream processing applications
- RO6.1: Analysis of memory consumption for parallel stream applications
- RO7: Evaluating matrix multiplication in limited-resource hardware
- RO8: Evaluating GPGPU parallelism for data processing applications

These ROs permit us to assess parallelism's programmability, performance, and energy consumption with different PPIs across two main parallelism models of data and stream. In the following Sections, we set the conditions for assessing each RO.

4.2 Hardware and Software Environment

We summarize the hardware configurations and kernel versions in Table 4.1 for the experimental phases. A common feature between all hardware is a 64 GB MicroSD card permanent storage with 104 MB/s. Other relevant software versions are GCC 9.4.0, Intel TBB(2020.1), and FastFlow (3.0.0). Additionally, we always use the -O3 compilation optimization flag to improve performance.

Table 4.1 – Low-resource hardware configurations and kernel version.

Device	CPU	CPU Freq (MHz)	GPU	RAM	Kernel
Jetson	Quad-core Cortex-A57	1479	128-core NVIDIA Maxwell	4 GB LPDDR4 SDRAM	4.9.253-tegra
Raspberry	Quad-core Cortex-A72	2200	Integrated	8 GB LPDDR4 SDRAM	5.4.0-1065-raspi
Odroid-N2	Quad-core Cortex-A73 & Dual-core Cortex-A53	1908 & 2208	Mali G52 GPU	4 GB LPDDR4 SDRAM	4.9.277-122-odroid

4.3 Applications

This Section explains the applications we use in this Master's thesis. To that end, Section 4.3.1 explains the stream processing applications, and Section 4.3.2 showcases the data parallelism applications.

4.3.1 Stream Processing

In this Master's thesis, we specifically focus on real-world stream processing applications that explore Pipeline parallelism. For our experimental analysis, we consider four real-world applications: Ferret, Bzip2, Lane Detection, and Face Recognition. Figure 4.1 depicts the execution data flow of all applications. We further summarize their main characteristics as follows:

- Ferret [15] is a content similarity search application for feature-rich data such as video, audio, and images. In this case, the version we use is an adaptation for image search. The parallel version has six stages. The first stage implements sequential input, and the final is sequential output. The four middle stages execute the bulk of the processing and are all parallel.
- Bzip2 [23] is a compression program in Linux-based distributions. It has independent modules for compression and decompression. The original Pthreads implementation models a three stages pipeline and has a handwritten reordering algorithm in the output filter to maintain the integrity of the compressed data.
- Lane Detection [31] reads an input video stream from a camera recording the whole lane in front of the vehicle. Then, it outputs the detected lanes after a sequence of filters. It utilizes the Canny filter and Hough transformations, which the OpenCV C++ image processing library provides. The parallel implementation technique uses a three stages pipeline where the first and last stages perform sequential I/O, and the central one collapses all processing filters in Parallel. The frame order must be guaranteed to keep the integrity of the video.
- Face Recognition [31] reads a video input stream from a camera positioned before a point of interest. Then, it detects faces and uses an image database to recognize if that face belongs to a specific person. This database is pre-processed and derives from a series of training images from that face of interest. The OpenCV image processing runtime also supports this application. The parallelism implementation consists of a pipeline with three stages where the middle stage is parallel, and the writer stage must also reorganize the output video frames before outputting them.

Beyond the data flow, these applications have different characteristics. Lane Detection is higher-throughput than the logically similar Face Recognition implementation. Bzip2 has high disk and memory usage, and the parallel version of Ferret typically spawns multiple threads that vastly oversubscribe the available number of cores in a system.

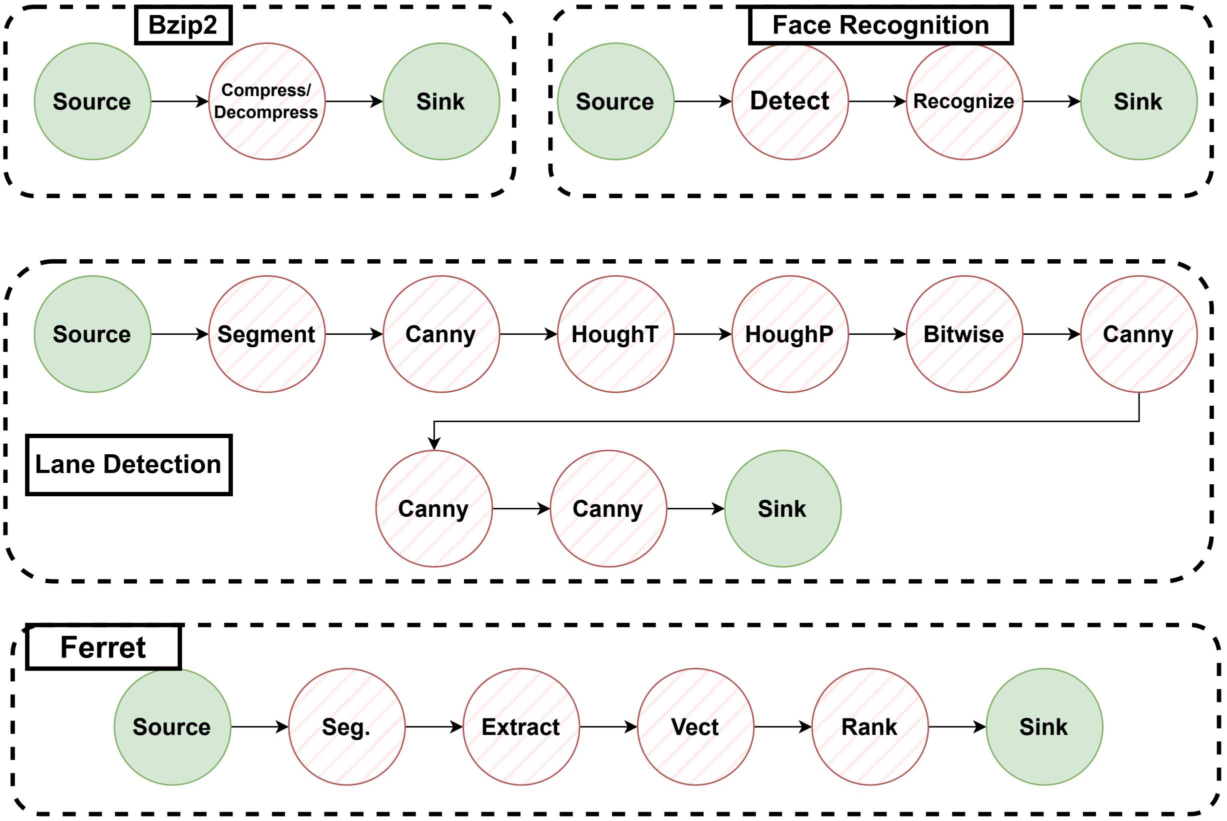


Figure 4.1 – Stream processing applications execution flow. *Solid green circles are sequential, and red crossed ones are parallel. Arrows indicate data dependencies.*

4.3.2 Data Parallelism

The goal of NAS Parallel benchmarks is to measure and compare parallel hardware objectively [12]. They are popular in the research community since they serve as a representative set of real-world workloads. Therefore, considerable research uses the NPB to evaluate their novel algorithms, optimization strategies, parallelism concepts, and many others [54].

NPB's applications come from the CFD (computational fluid dynamics) field. The benchmark suite contains five kernels representing core mathematical methods and three pseudo-applications using similar operations to represent closer to real-world CFD applications. Each application poses different computational characteristics, such as irregular memory accesses, complex data dependencies, and intensive data communications. We briefly describe these kernels below:

- **Embarrassingly Parallel (EP)** computes pairs of Gaussian random deviates according to a specific scheme. EP is useful to measure the peak floating-point operation performance of a given platform [12].

- **Multi Grid (MG)** uses a V-cycle MultiGrid method to compute the solution of the 3-D scalar Poisson equation while continuously alternating between fine and coarse grids. MG tests both short and long-distance data movement[44].

- **Conjugate Gradient (CG)** approximates the smallest eigenvalue of a large, sparse, unstructured matrix. CG tests data communication mechanisms, memory locality, and cache [44].

- **Discrete 3D Fast Fourier Transform (FT)** computes a Fast Fourier Transform of a 3D partial differential equation in an iterative loop. FT simulates intensive long-distance communications [12].

- **Integer Sort (IS)** sorts integer values from a sparse set with a bucket-sorting algorithm. IS measures integer computations and data movements [44].

We fully experiment and test with **Block Tri-diagonal solver (BT)**, **Scalar Penta-diagonal solver (SP)**, and **Lower-Upper Gauss-Seidel solver (LU)**. In summary, they solve three sets of uncoupled systems of partial differential equations that describe a flow of incomprehensible fluids. They stress multiple features of the target architecture and are closer to real-world applications since they combine different computational methods.

The NPB has different workload sizes that express classes. Classes S and W are suitable for quick and small tests; classes A, B, and C are the standard tests, where the complexity of each one increases four times between one class and the next. Classes D, E, and F are for heavy tests, with a 16 times complexity increase between classes.

The illustration in Figure 4.2 displays the execution flow of NPB. Solid-lined rounded rectangles represent these programs' functions, while routines are dotted rectangles. Routines are a series of functions. The function that consumes the most computation time within each benchmark is highlighted with blue dashed borders. Meanwhile, the parallel functions are solid green.

Each benchmark starts with an initialization step and concludes with a verification procedure that checks the correctness of the results. In between, the benchmark performs its primary computations as part of an iterative process. EP benchmark is an exception since its most time-consuming function does not fall within an iteration logic. Furthermore, EP and IS are the only benchmarks with a single primary function. The measurement of the total execution time begins immediately after the initialization step and terminates immediately at the start of the verification function.

All kernels and applications exhibit various features that pose significant challenges for parallelization. These features include tasks with non-uniform computation, multi-dimensional arrays with up to five dimensions requiring distinct data access patterns, data dependencies requiring strategies to eliminate or minimize performance degradation, and large data chunks. From now on, we refer to the NPB set of kernels and pseudo-

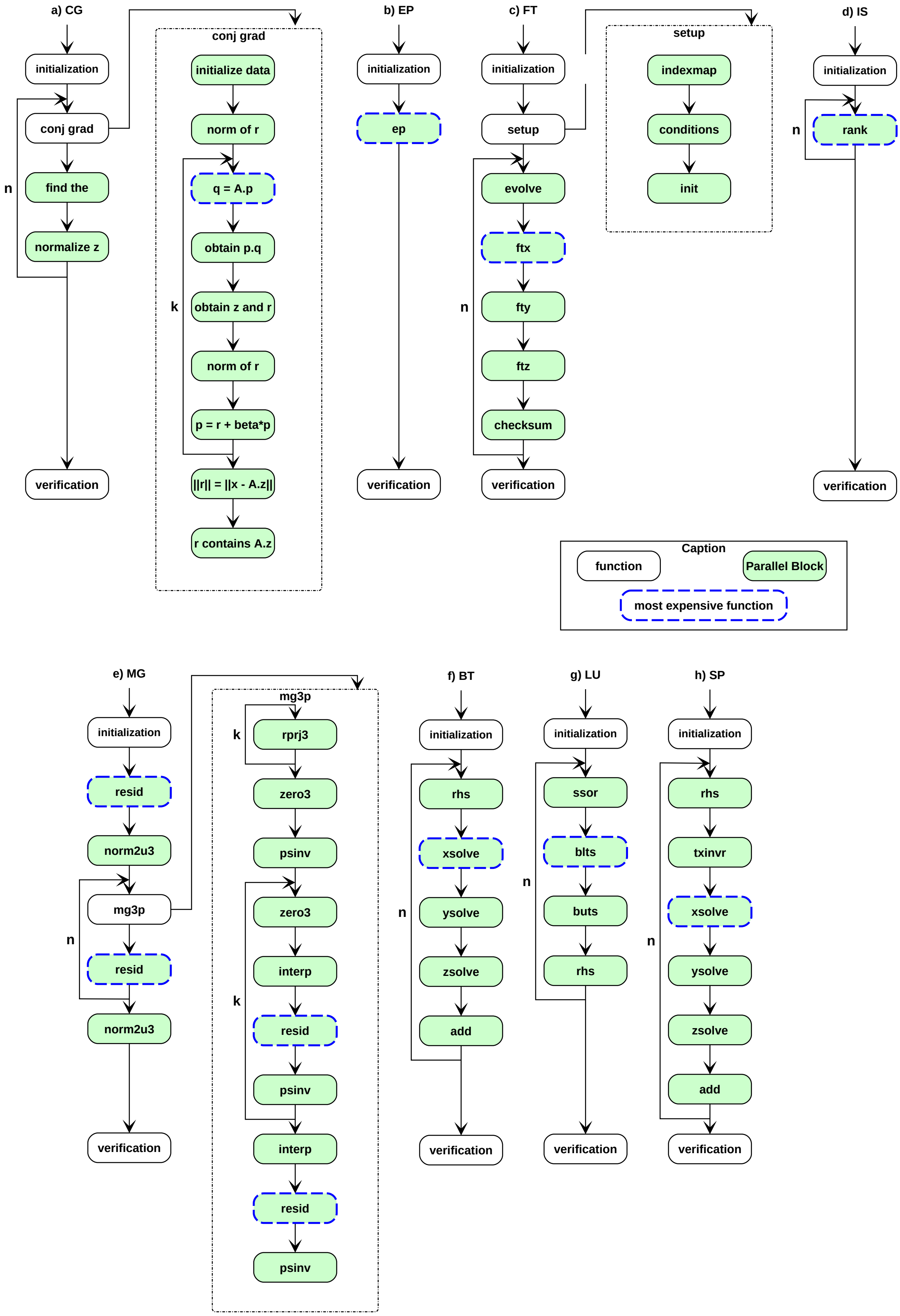


Figure 4.2 – Data processing applications execution flow. *Extracted from [9].*

applications simply as applications. All applications are parallelized with a variation of the MapReduce pattern, as is typical in most data processing applications. We use an existing parallel implementation available in OpenMP, TBB, and Fastflow [54], proven to perform nearly identically to the original Fortran version. Additionally, the CUDA versions of NPB are proven to perform well on HPC GPGPUs [9].

4.4 Metrics

This Section explains the metrics we use to measure the efficiency and effectiveness of the parallel computing features derived from our research questions. We explain the performance metrics in Section 4.4.1 and the consumption metrics in Section 4.4.2. Finally, we also explain programmability metrics in Section4.4.3

4.4.1 Performance

For performance, we measure latency, throughput, and execution time. **Latency** represents the time a data element takes to reach its destination or target processing element; in other words, latency is the response time between an event generation and the completion of its computation. **Throughput** represents the average number of data elements completed per second. **Execution time** represents the total time in seconds it takes for the application to execute from start to finish. All performance metrics derive from time measurements taken within the source code using the standard C++ Chrono library.

Even though latency and execution time are related, they might not be directly correlated. For instance, potentially to the detriment of the other metric, real-time systems require close-to-zero latency while training machine learning models would strive for a shorter execution time. Conversely, throughput is a metric most relevant to stream processing systems that operate based on time windows.

4.4.2 Consumption

For consumption metrics, we measure energy, power, and memory. **Energy** is the accumulated total amount of Watt-hour consumed from start to finish of the test period. **Power** is the average watts consumed from start to finish of any test period; we measure it by summing all instant readings from start to finish of the execution and then dividing by the number of samples. **Memory** is the highest consumption of MB of RAM observed at any point from the start to finish of the test period. The test period is the execution time.

Energy consumption in computer science is a critical financial and environmental challenge. It should be the lowest possible to reduce costs. Power dissipation measures the system's instant current and voltage demand. Ideal power dissipation should be consistently low and free from sudden increases in power demands. We monitor power peaks to detect potential surges. RAM consumption is also essential to assess the impact of parallel programming and its viability on limited-resource hardware. However, if the system has excess memory, its significance decreases, although it should ideally be as low as possible.

Some limited-resource hardware includes counters for energy consumption, but this capability is only sometimes present. Some energy measurement systems estimate energy consumption based on hardware counters instead of actively measuring it [84]. A standard method to measure energy consumption is to use a voltmeter and ammeter by connecting them between the device and its power source. Our research uses the UM25C [45] USB device to measure energy consumption. The UM25C transmits its measurements via Bluetooth, which we collect using an independent computer and log as a CSV file. Later, we correlate the energy consumption readings with execution logs to determine the consumption in Wh.

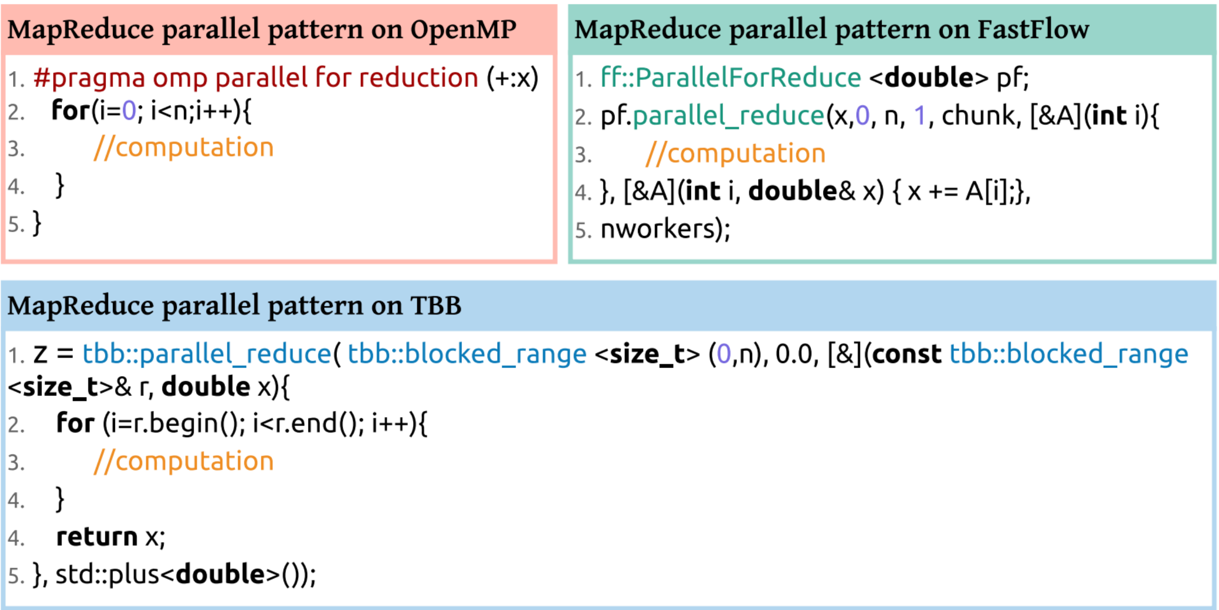
We sample energy consumption measurements every 0.3 seconds, the highest sampling rate we can obtain from the UM25C device. Since we correlate execution time logs with energy consumption logs, we must round execution timestamps to the closest match. Therefore, we may overestimate or underestimate the energy consumption by up to 0.3 seconds. In this Master's thesis experiment, the energy consumption approximation never entails roundings greater than 0.002 mWh: 0.001 mWh for the start and potentially other 0.001 mWh for the end.

4.4.3 Programming Productivity

Many studies in the parallel programming domain evaluate technical factors such as execution time, speedup, memory, and energy consumption. Programming productivity is another crucial factor in comparing different parallel programming interfaces. Developing more productive PPIs and refining existing ones from usability evaluation is possible. However, this is a complex task since it involves human beings. Therefore, many researchers use established code metrics to approximate productivity evaluation.

It is crucial to evaluate and compare PPMs concerning productivity. For that purpose, it is possible to use code metrics based on: code size, for example, Source Lines of Code (SLOC), Number of Characters, and Tokens of Code; complexity evaluation, for example, Cyclomatic Complexity Number and Information Flow Complexity; and development effort, for example, Halstead, and Constructive Cost Model. While helpful, code size and complexity evaluations cannot predict the effort required to develop a parallel application [7]. Evaluations that estimate development efforts are more helpful but still have limitations. In this Master's thesis, we measure SLOC and Halstead.

SLOC (source lines of code) does not count blank or commented lines. It has limitations since it does not consider software complexity, code structure, number of functions, or connections between modules. SLOC primarily measures verbosity, so comparing different versions of the same program in the same programming language is better. We can observe this is an issue taking the example of Figure 4.3, where we have the same algorithm, expressed in the same number of lines, but using different PPIs with potentially different complexities involved.



MapReduce parallel pattern on OpenMP

```
#pragma omp parallel for reduction (+:x)
  for(i=0; i<n;i++){
      //computation
  }
}
```

MapReduce parallel pattern on FastFlow

```
ff::ParallelForReduce <double> pf;
pf.parallel_reduce(x,0, n, 1, chunk, [&A](int i){
      //computation
}, [&A](int i, double& x) { x += A[i];},
nworkers);
```

MapReduce parallel pattern on TBB

```
z = tbb::parallel_reduce( tbb::blocked_range <size_t> (0,n), 0.0, [&](const tbb::blocked_range
<size_t>& r, double x){
   for (i=r.begin(); i<r.end(); i++){
      //computation
   }
   return x;
}, std::plus<double>());
```

Figure 4.3 – Example extracted from [54].

Halstead metrics quantify software complexity. Halstead metrics measure the program length, vocabulary size, volume, and difficulty estimating the development effort. The central idea is that the program's complexity is related to the number of unique operations and operands used in the program and the number of times they are used. The higher the Halstead metrics, the higher the program complexity. Halstead metrics might not, however, represent ease of programmability. In summary, the Halstead metrics provide a quantitative way to assess the complexity of a software system based on the number and diversity of its constituent parts. To measure Halstead, we adapted the Parallel Halstead algorithm [7] to include OpenMP and Threads.

4.5 Parallel Programming Interfaces

The parallel programming interfaces we use for our experimental analysis have different characteristics. They are SPar, FastFlow, TBB, OpenMP, and Threads. Additionally, for GPGPU, we also experiment with CUDA and OpenCL. They are different from one another because they have different design goals and parallelism implementation strategies. Even if we can create a logically similar parallel implementation of an application using them, they can still have several different implications when executing. Furthermore, they may have different abstraction and programmability features. In the next Chapter, we explain them in greater detail, where the first two research questions explore their main differences.

4.6 Execution and Graphs

This Section aims to provide details about the execution parameters from all the experimental data gathered and summarized in this Master's thesis. We also outline some of the main graph configurations.

We repeated all experiments five times. From that, we calculate the median average and the standard deviation. We use these results as the data for our graph plots. Concerning the graphs, the $X-axis$ is the degree of parallelism. The degree of parallelism ranges from 1 to the maximum number of cores available in each hardware configuration. That is one up to 4 for Jetson and Raspberry and one up to 6 for Odroid. An important aspect to consider is that the degree of parallelism may not represent the actual active thread count in the system. Instead, each runtime may spawn one dedicated thread for each stage, or in the case of parallel stages, a thread pool is determined in size by the degree of parallelism.

In all graphs, the 0 value in the $X - axis$ always represents the sequential execution. Furthermore, in the graphs, the $Y - axis$ represents one of the performance or consumption metrics. Finally, graphs have a logarithmic scale of 2, and we show error bars to represent the standard deviation, which in some cases may not be visible due to its low value. Lastly, we guarantee the output correctness of the parallel versions by comparing the resulting output MD5 (Message-Digest algorithm 5) checksum with the original sequential version.

Another point to consider is when we refer to the text about percentage differences. We calculate *percentage differences* using the absolute difference between two values, divide by the average, and multiply by 100 to get the percentage; we use the following formula $(|value - referencevalue|/average) * 100$. When there are multiple values, the reference value is the sequential version.

4.6.1 Input Data

- **Lane Detection** 's input is a video of a truck driving on the road. It is an mp4 video with 640×360 resolution with 30 frames per second (H.264 codec). The video lasts 3 seconds and has a file size of 0.57 MB.
- **Bzip2** 's input is a dump file from the Wikipedia database with a total size of 349.1 MB. Bzip2 Storm/Flink has used as input a random characters text file with 51 MB size.
- **Face Recognition** 's input consists of pictures of former US president Obama's face plus a video recorded during a talk that contains images of his face and the faces of other people in the audience at some points. The training phase of the application uses this set of pictures, while the recognition of the former president's face uses video. It is an mp4 video with 640×360 resolution with 30 frames per second (H.264 codec). The video lasts 3 seconds and has a file size of 0.57 MB.
- **Ferret**'s input is a set of images from the *large* alias in the Parsec benchmark [15]. It performs 256 image similarity queries from a 34,973 image bank. All images are 128x96 resolution JPEG files. Each query searches for the top $K = 10$ most similar images.

NPB's workload is class B for the five kernels (EP, CG, MG, FT, and IS) and class A for the three applications (BT, SP, and LU). Class B is the largest workload we could securely execute every test with 4 GB of RAM. Furthermore, we use class A for the applications since they take longer to process. Specifically for NPB, we change the benchmark after each execution to ensure the system does not pre-load data into the cache and reuse it between executions.

5. RESEARCH OBJECTIVES’ ANALYSIS

This Chapter presents an analysis of this Master’s thesis research objectives (Section 4.1). To help readers navigate ROs, we categorize them in Table 5.1 using the characteristics we observe in Chapter 4. In Table 5.1, the application type describes if the RO studies data or stream processing applications. The analysis describes the metrics we use in each RO, while the devices describe the hardware we use in that analysis. Parallelism refers to CPU multithreading or GPGPU accelerators.

Table 5.1 – Classification of Research Objectives.

Research Objectives	Application Type	Analysis	Devices	Parallelism
1	Data & Stream	Programmability	-	All
2	Data & Stream	Programmability	-	All
3	Data	Performance	All	CPU
3.1	Data	Performance	All	CPU
3.2	Data	Performance	All	CPU
3.3	Data	Performance	All	CPU
3.4	Data	Performance	All	CPU
3.5	Data	Performance	All	CPU
3.6	Data	Performance	All	CPU
3.7	Data	Performance	All	CPU
4	Data	Consumption	All	CPU
5	Stream	Performance	All	CPU
5.1	Stream	Performance	All	CPU
5.2	Stream	Performance	All	CPU
6	Stream	Consumption	All	CPU
6.1	Stream	Consumption	All	CPU
7	Data	Cons. & Perf.	Jetson & Odroid	GPGPU & CPU
8	Data	Cons. & Perf.	Jetson	GPGPU

We also highlight the experimental methodology using a flowchart that results in each research objective in Figure 5.1. It connects parallelism versions, applications, hardware, metrics, and the research objective. The numbers on the edges represent the accumulated number of results or experiments that each RO considers. The origin of the edge indicates whether it encompasses one (dotted line origin) or multiple features (straight line origin). Sometimes, we use colours to create a joint group of experiments, facilitating their origin; otherwise, they encompass everything that arrives in them. One final consideration is that ROs 7 and 8 are exceptional cases, and we use dotted edges.

Each following Section describes one RO. In them, we explain the context, forces, discussion, and threats to the validity of our analysis. The **Context -** summarizes the RO’s motivation and the challenge it brings. The **Forces -** associated with the RO explain factors in play and possible solutions. For the **Discussion -**, we show collected results, findings, and evaluations. We also explain the **threats to the validity** or **limitations** of our analysis for the leading ROs.

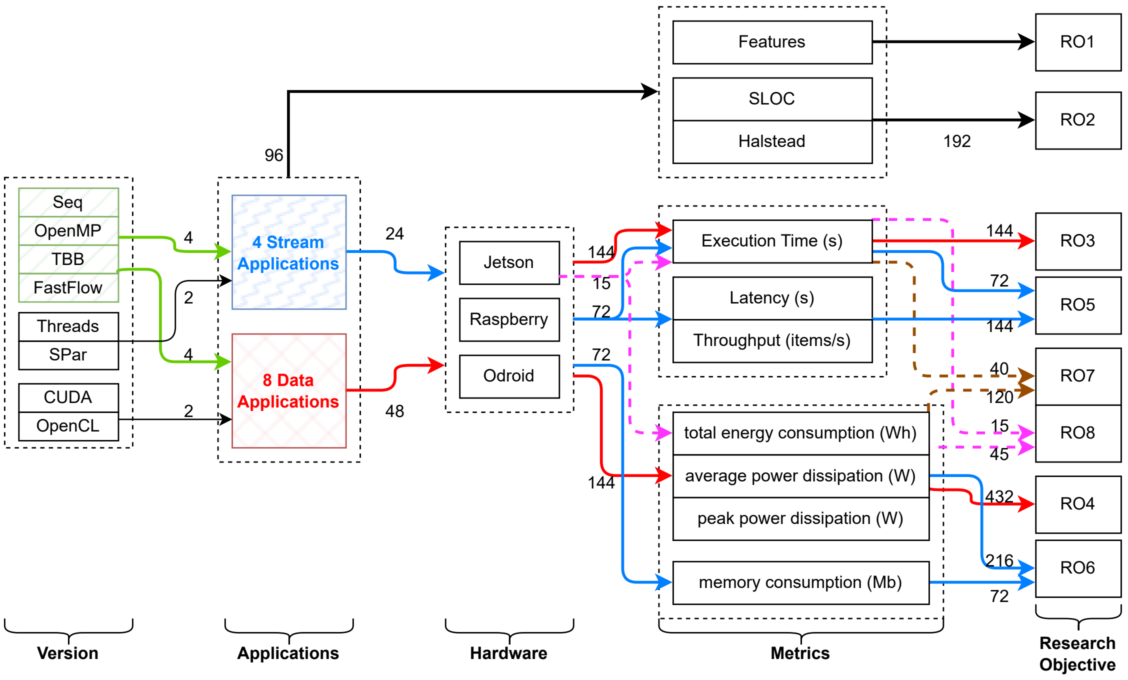


Figure 5.1 – Experimental methodology using the research objectives.

5.1 RO1: Evaluating programming features for the parallel programming interfaces

Context and Forces - For limited-resource hardware, developers typically must choose low-abstraction solutions that allow expressive control of the available hardware[13]. For this research question, we assess the most relevant PPI features, limitations, and challenges associated with parallelism in limited-resource hardware.

Discussion - For shared-memory multithreading, we highlight the main differences between each evaluated PPI in Table 5.2. We explain the characteristics from the Table during this discussion session. One consideration is balancing the application computation between the fewer available processors. Unlike x86 architectures, our devices do not have Simultaneous Multithreading (SMT or Hyper-threading). Therefore, the programmer can conFigure parallel threads to oversubscribe the available cores, use the maximum amount, or even underutilize available processing cores (valid as long as memory contention is the main issue). The number of parallel processing threads per processing stage is user-defined in OpenMP, FastFlow, SPar, and Threads; otherwise, they use maximum resources. TBB is the only interface that automatically chooses these values dynamically according to its scheduler logic. TBB allows users to set the maximum number of threads their scheduler can use.

Another factor to consider is mapping threads into physical cores. The operational systems are typically responsible, but some interfaces, such as FF, have custom thread mapping algorithms. However, Threads allows pinning threads to physical cores

Table 5.2 – Parallel Programming Interfaces main characteristics comparison.

PPI	Programming Model	Communication Pattern	Thread Mapping	Degree of Parallelism	Memory Config	Blocking Mode	Stream Support	Data Support
SPar	DSL	Runtime	Yes	User defined	Queue size	Yes	Yes	Partial
FastFlow	Patterns	Static buffer	Yes	User defined	Queue size	Yes	Yes	Yes
TBB	Patterns	Work-stealing	No	Dynamic	Number of Tokens	No	Yes	Yes
OpenMP	Compiler Directives	Static buffer (data) User-defined (stream)	No	User defined	Queue size	Yes	No	Yes
Threads	Explicit	User defined	Explicit	User defined	Fully	Explicit	User defined	User defined

to optimize cache coherency. OpenMP and TBB do not directly provide this functionality. For all NPB applications using FF, we must manually define the cache-line size for cache padding, an automatic process on TBB and OpenMP.

Threads and exclusively stream processing OpenMP are the only interfaces that require the developer to directly specify data scheduling or load balancing algorithms, which may be a complex implementation [39]. In Table 5.2, we refer to it as Communication Pattern. For this data scheduling algorithm, one can consider busy waiting; this is fully controlled by the programmer using Threads, which is impossible in TBB. FastFlow, SPar, and OpenMP allows using this configuration by simply enabling it with the *omp_wait_policy* compilation flag.

One other crucial factor for some limited-resource hardware is using static memory consumption. Within the minimum required range by each application, we observe that all parallelism interfaces allocate up to a maximum amount of required memory. FastFlow, SPar, and OpenMP stream only allow the user to define the size of internal communication buffer queues. TBB allocates memory dynamically; its pipeline implementation permits defining the number of items (tokens) alive in the stream, which is the only configurable parameter that affects the PPI memory consumption. Threads is fully controlled by the user's implementation.

Evaluating the available programming features, OpenMP is the only PPI that provides explicit SIMD parallelism options. For GPGUs parallel programming, the developers must take into account different concerns. Reducing the memory footprint is essential in limited-resource hardware, which requires changing the algorithms to use smaller data types, avoiding duplicating data, and possibly separating the original algorithm into smaller parts that can fit in the GPGPU. Another vital consideration is available data types. Mali G52 only supports 32-bit floating point numbers, which does not meet the NPB applications' 64-bit precision requirements.

Another critical change for GPGPU is reducing the computational kernel complexity; this involves minimizing the number of instructions and memory accesses during each kernel invocation. Even with enough available memory, threads per block, and blocks available in the GPGPU, HPC servers' functional GPGPU code may not execute correctly on limited-resource GPGPUs because of high kernel complexity. In other words, if there are not enough registers to invoke the thread blocks, an error is silently generated during the execution. This scenario is the case for some NPB applications on the Jetson board. Choosing the appropriate parallelism interface for GPGPU is also a limited choice because this hardware frequently needs to provide drivers for GPGPU interfaces such as OpenACC, CUDA, or even OpenCL. In Jetson, we can only execute CUDA code while only OpenCL in Odroid.

SPBench (stream processing benchmark) applications did not require any changes regarding the original implementation (see Section 4.3) to run in limited-resource hardware. However, some PPI characteristics are not ideal for this type of hardware, as we examine in the following ROs of this Chapter. For NPB applications, only FastFlow requires manually setting the cache-line size. Otherwise, the parallel code is the same as executed in HPC servers [54].

The main **limitation** is the set of stream and data processing applications we assess. In developing other parallel applications, additional features we did not consider may be necessary. These extra features may further vary development decision-making between interfaces.

5.2 RO2: Evaluating programmability metrics for the parallel programming interfaces

Context - When choosing a parallel programming interface, developers find several alternatives. Beyond efficiency, each one may have a different learning curve, expressiveness, verbosity, ease of use, portability, and level of abstraction. For instance, according to their design choices, some interfaces may favor generalization, abstractions over performance, and customization range over simplicity. This research question assesses the programmability or productivity of higher-level abstractions in limited-resource hardware. We also discuss the most relevant changes, limitations, and challenges to porting the code from typical HPC servers to limited-resource hardware.

Forces - SPar is a high-level stream processing domain-specific language. It uses code annotations that do not require code-refactoring and low-level implementation. It typically requires a few lines to enable parallelism but is not customizable. OpenMP is an unstructured approach that leverages pre-compiler directives and library functions to enable parallelism. The exception is for stream processing applications, requiring additional mechanisms to function correctly [39].

TBB and FastFlow adopt the structured parallel programming approach. Programmers must structure their code into pre-defined templates such as MapReduce or Pipeline. On the other hand, TBB is not very customizable, whereas FastFlow allows a much greater level of customization and adaptation of the background parallelism behavior. Threads require the programmer to define every aspect of the parallelism algorithm using a set of functions that interact with the operational system.

Discussion - Regarding stream processing applications in Figure 5.2, the average source lines of code for all four applications combined are 19.75 for Seq, 73.25 for Fastflow, 71.50 for TBB, 210.75 for OpenMP, 194.50 for Threads, and 38.75 for SPar. This metric primarily showcases the level of code intrusion each parallelism interface requires to model stream parallelism concerning the sequential version. The Seq SLOC count is lower because SPBench exposes a very concise code, which is the base for every other parallelism implementation. The graph shows patterns: the implementation with SPar uses the lowest; OpenMP and Threads are similar and use the highest; FastFlow and TBB are in the middle but closer to SPar.

SPar achieves low lines of code since it is a high-level domain-specific language for stream processing. TBB and FastFlow use the same structured parallel programming approach, which requires code re-structuring. Their runtime manages the most complicated aspects of parallel programming. On the other hand, Threads require the full implementation of all parallelism features. The outliner is OpenMP because it typically only requires a few pragma notations to enable parallelism; this is not true for stream processing applications, and it requires several different mechanisms to model pipeline stream parallelism [39]. At its highest, OpenMP has 858.33% more SLOC than SPar.

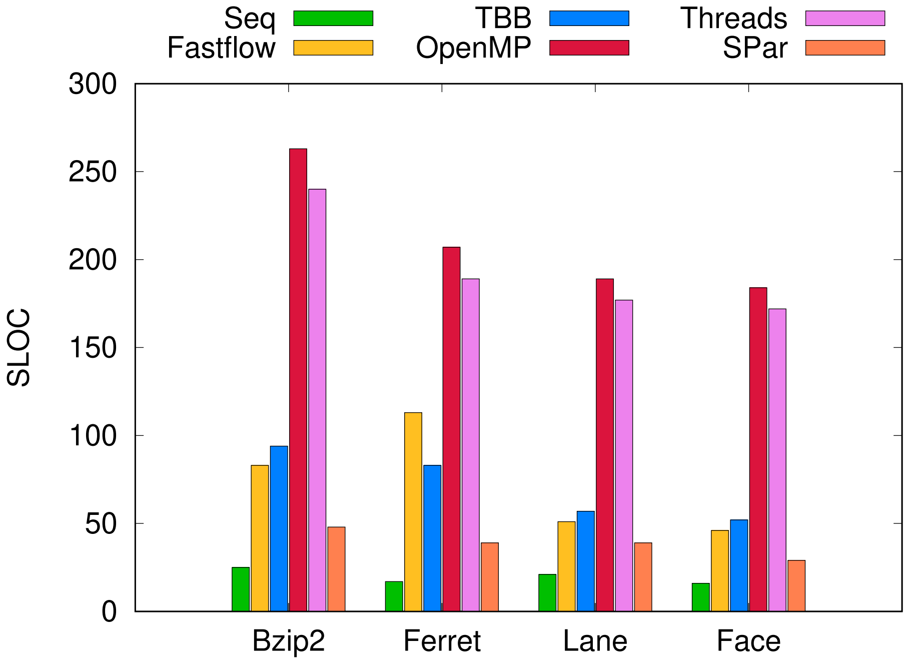


Figure 5.2 – SPBench applications SLOC.

For NPB data processing applications in Figure 5.3, the average SLOC for all eight applications combined is 1237 for Seq, 1307.63 for Fastflow, 1328.63 for TBB, and 1303.75 for OpenMP. Unlike the previous graph, the sequential values here are much higher because they include the entire source code. All parallel implementations also use the same sequential base code. Furthermore, unlike the previous applications, the differences between TBB, OpenMP, and FastFlow are much smaller. The highest difference is on EP, where TBB has 7.10% more SLOC than OpenMP. For this class of applications, the difference in SLOC between each parallel programming model is insignificant. In this case, OpenMP provides enough support to parallelize the entire code using its pragma directives. TBB and FastFlow provide a relatively concise MapReduce pattern that can efficiently leverage parallelism.

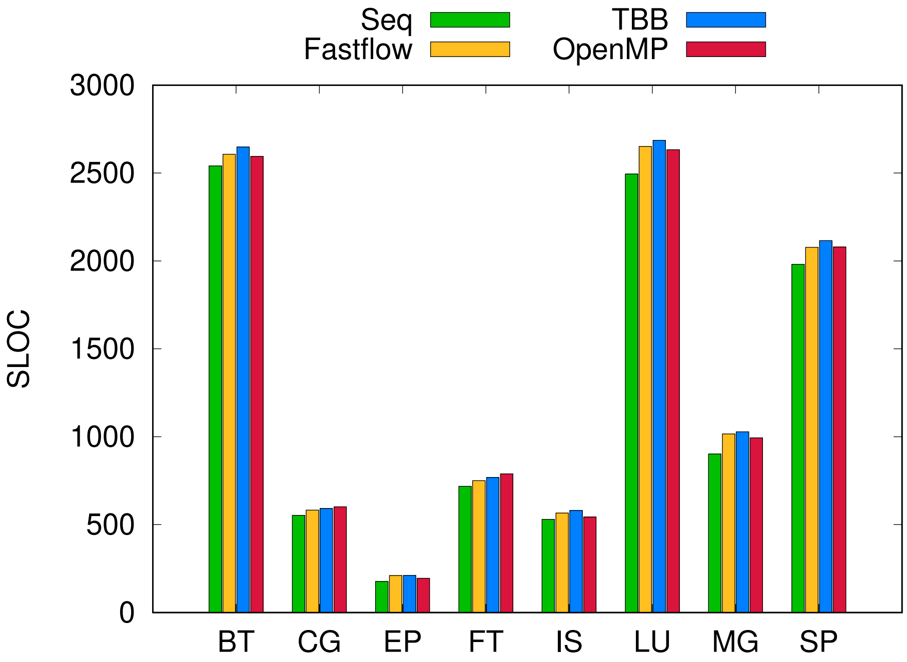


Figure 5.3 – NPB applications SLOC.

Moving to stream processing applications effort evaluation, Table 5.3 showcases the Halstead estimated development effort normalized between 0 and 1, where 0 is the sequential and 1 is the highest Halstead measurement. Halstead measurements follow a similar pattern to SLOC, where SPar is the lowest, OpenMP and Threads are higher, and TBB and FastFlow are in the middle. However, Halstead indicates a more significant gap between SPar and TBB/FastFlow. This is because SPar can preserve the original code structure. We also see that OpenMP and Threads have similar estimated development efforts. After all, they both require explicit communication and synchronization logic, while all others do not.

Table 5.4 showcases the Halstead measurements for all 8 NPB data processing applications. The values are also normalized. Compared to SLOC, Halstead effort estimation reveals a different insight: OpenMP is, on average, significantly lower effort than FastFlow and TBB. That is because its pragma directives are suitable for data processing applications, where they can correctly model all encountered parallelism features without changing much of the original code. Indeed, it still requires a great rationale about the

Table 5.3 – Normalized Halstead for SPBench applications.

App	Fastflow	TBB	OpenMP	Threads	SPar
Bzip2	0.32	0.31	0.96	1.00	0.04
Ferret	0.79	0.49	0.98	1.00	0.06
Lane	0.32	0.34	0.97	1.00	0.05
Face	0.24	0.25	0.97	1.00	0.04
Total Avg.	0.41	0.35	0.97	1.00	0.05

parallel logic, but they are an efficient tool for these applications. `FastFlow` and TBB have a similar Halstead evaluation for all applications except EP, where TBB is estimated to be a lower effort. That is due to a design choice where `FastFlow` implements a MapReduce pattern, which requires defining the reduce operation. In contrast, TBB uses a Map pattern with atomic operations to reduce.

Table 5.4 – Normalized Halstead for NPB applications. Normalized values.

App	FastFlow	TBB	OpenMP
BT	0.98	1.00	0.80
CG	1.00	0.92	0.74
EP	1.00	0.77	0.58
FT	0.92	1.00	0.75
IS	0.92	1.00	0.59
LU	0.99	1.00	0.73
MG	0.95	1.00	0.58
SP	0.96	1.00	0.80
Total Avg.	0.96	0.96	0.70

The main **threat to validity** of this analysis is the low number of metrics considered. As this RO showed, SLOC and Halstead yield different insights. We might obtain new information by using additional metrics to differentiate the parallelism interfaces [7, 6]. Ultimately, programming efficiency may also be subjective since it involves humans.

5.3 RO3: Evaluating performance metrics for data processing applications

Context - Parallel data processing plays a critical role in computer science by enabling faster and more efficient processing of large amounts of data. This data is available at the start of the computation and is divided among the parallel computing elements. It is possible to measure the performance of data processing applications by measuring the difference between the time of starting and completing any computation. It ideally completes in the shortest period.

Forces - The performance of data processing applications is heavily influenced by the application logic and available hardware resources. Given that such applications process large volumes of data, the choice of hardware configuration and their inherent architectural characteristics significantly impact their performance. In addition, parallel programming interfaces (PPIs) are crucial in selecting the appropriate parallelism algorithms to meet these applications' performance requirements. In this context, OpenMP is the de-facto standard for multi-core shared-memory architectures. It has been shown to deliver superior performance and is widely recognized in the HPC community [54].

Threats to the validity This RO and its derived ROs have some limitations. First, we only use NPB applications and therefore are limited to the computational characteristics that, although varied, they provide. For instance, although we have eight applications, some are structurally similar such as SP and BT. Another point of consideration is that we cannot use system profilers on Odroid and Jetson devices because they are not available. Instead, we must rely on simulation tools for these devices, such as Valgrind. Conversely, Raspberry allows us to accurately count hardware events with Linux *perf* tools. Finally, another consideration, heavily related to RO 3.7, is that Raspberry does not dissipate heat as efficiently as the other devices and might have some issues with higher degrees of parallelism.

Discussion - We showcase all NPB data processing applications' execution time measurements in Table 5.5. In this case, we demonstrate the results for Jetson, Raspberry, and Odroid devices considering all 8 NPB applications. All parallel versions use the maximum available hardware resources. We also display the total average amount of all devices and applications for each version at the bottom of the table. We highlight the best parallel performance PPI in light green and the worst in light red. Compared with the parallelism gains, OpenMP is best; it is 0.7% and 1.46% better than TBB and FastFlow, respectively. However, this is a generalized result that has a significant standard deviation; there are more fine-grained details we should consider.

Overall, Table 5.5 shows that the performance varies depending on the application, parallel processing technique, and device. For instance, comparing FastFlow with OpenMP on the BT application, we see that OpenMP is significantly better in Jetson and Raspberry while it is similar in Odroid. These differences appear in more applications, which we investigate individually in the remainder of this Section and derived ROs.

5.3.1 RO3.1: Analysis of MapReduce with no data dependencies

EP implements a MapReduce strategy with no data dependencies. With an efficient parallelism implementation, it has the most significant scalability potential. In Figure 5.4, Jetson with a degree of parallelism 2 has a nearly linear 1.95 speedup and 3.9 with 4. Odroid shows linear speedup using the four most efficient BIG.little cores but also benefits from the two low-performance cores, which increase the speedup from 4 to 5.15 with degrees of parallelism 4 and 6, respectively. We also see that TBB's dynamic schedul-

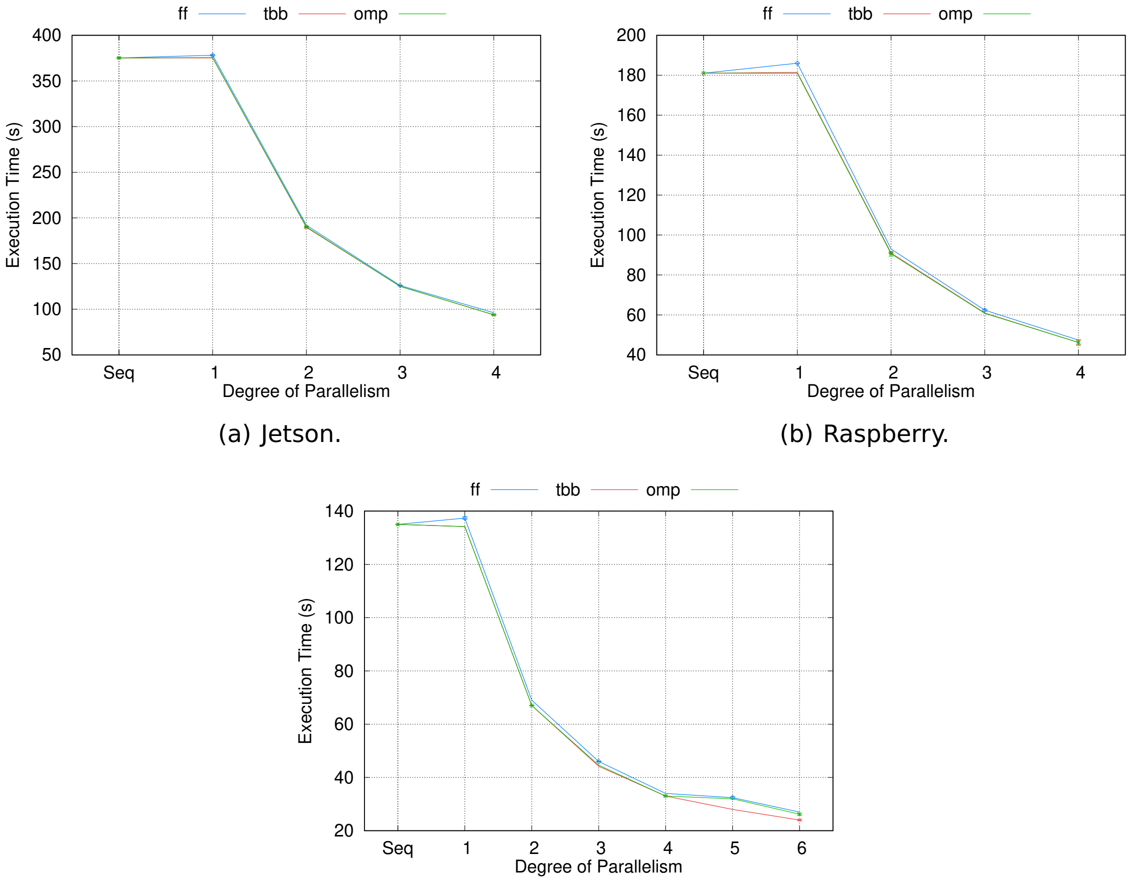
Table 5.5 – Execution time for NPB applications. For parallel versions, green and red highlight the best and worst observed results, respectively.

Device	Application	Execution Time (seconds)			
		Seq	FastFlow	OpenMP	TBB
Jetson	BT	172.60	54.00	50.00	54.00
	CG	403.00	164.40	136.80	163.80
	EP	375.20	96.20	94.00	93.80
	FT	282.00	83.40	82.40	82.40
	IS	8.00	2.00	2.00	2.00
	LU	149.60	51.20	51.40	51.00
	MG	18.80	9.00	9.00	9.00
	SP	122.00	53.40	52.80	52.80
Raspberry	BT	118.80	75.80	65.20	69.40
	CG	262.40	163.80	161.80	160.40
	EP	181.00	47.40	46.20	46.20
	FT	315.60	126.20	126.80	125.00
	IS	5.00	2.00	2.00	2.00
	LU	145.00	133.40	132.80	133.60
	MG	27.00	21.00	21.00	21.00
	SP	116.20	104.60	107.00	109.80
Odroid	BT	100.00	36.00	36.00	34.00
	CG	181.60	69.20	66.00	66.40
	EP	135.00	27.00	26.20	24.00
	FT	248.40	93.60	92.20	87.40
	IS	2.00	1.00	1.00	1.00
	LU	90.00	60.00	60.80	59.40
	MG	14.00	8.00	8.00	8.00
	SP	71.40	57.40	56.80	56.80
Total Avg.		147.69	64.17	62.01	63.05

ing can handle the BIG.little differences slightly better than FF's and OpenMP's static load scheduling. These findings align with the conclusion of past research [18]. EP is an application that achieves close to 100% constant CPU utilization. Raspberry shows 1.9 speedups with degree 2 and only 3.92 with degree 4. Therefore, low-resource hardware parallelism can increase speedup close to the ideal efficiency.

5.3.2 RO3.2: Evaluating Map with memory contention

The FT application contains three independent symmetric Fast Fourier Transform routines that compute three dimensions using a Map algorithm. However, this algorithm has many communications because it must decompose slices of the main 3D matrix into a 1D local array each time it applies an FFT resolution. Then it requires copying the results back. Data communication imposes a significant challenge to parallelism scalability.



(a) Jetson.

(b) Raspberry.

(c) Odroid.

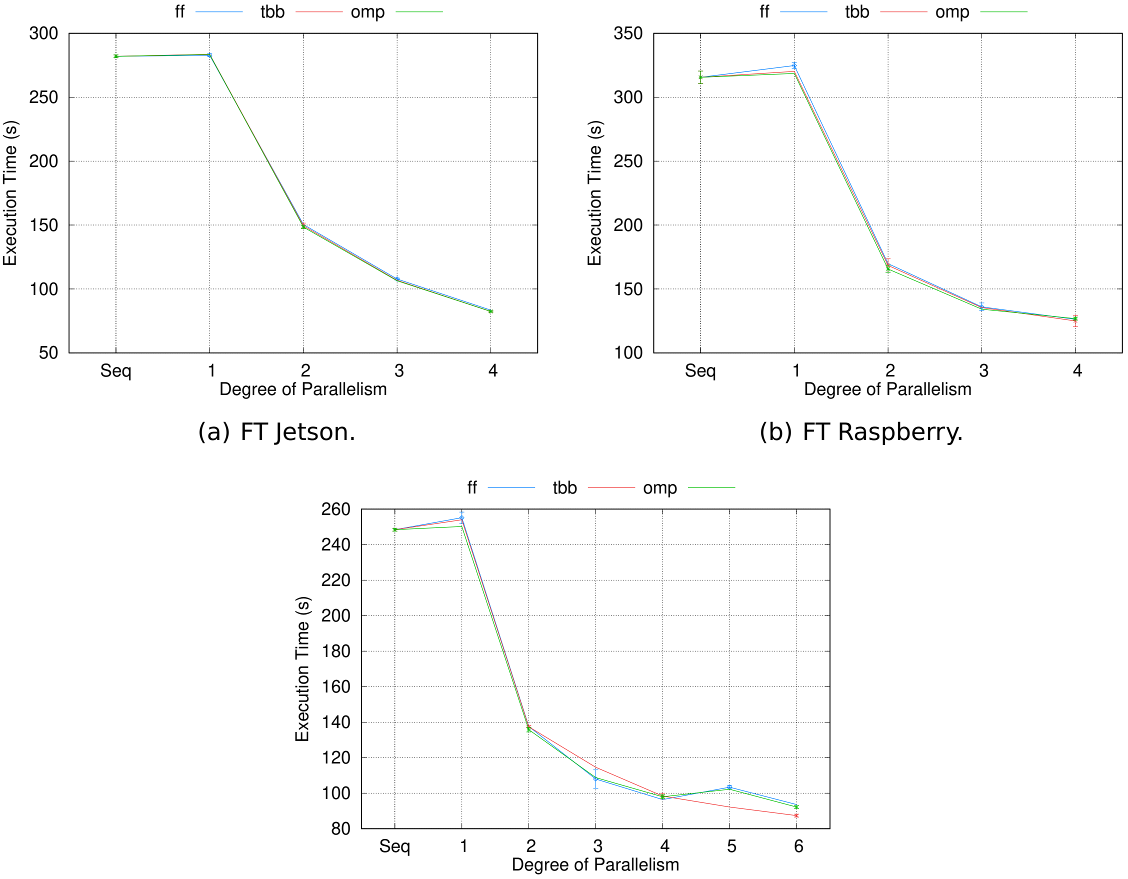
Figure 5.4 – EP application on multiple devices.

Therefore, FT can pressure the memory hierarchy mechanisms and internal data paths. This phenomenon is known as the "memory wall" or memory contention problem in performance scalability.

Figure 5.5 showcases the performance results of the FT application across all devices. We can compare it with EP results and observe lower scalability, even though the main computation loops have no data dependencies between each other. On Odroid, using maximum hardware resources, EP achieves a speedup of 5.1527 and FT 2.6920. On Raspberry, EP achieves 3.92 and FT 2.4883 speedups. Even with the memory contention problem, using parallelism improves performance.

5.3.3 RO3.3: Evaluating data locality on Maps

MG and CG parallelism applies multiple Map algorithms, while CG also applies multiple MapReduce algorithms. Furthermore, they both must perform multiple barrier logic synchronizations. In addition, another challenge is that both CG and MG display



(a) FT Jetson.

(b) FT Raspberry.

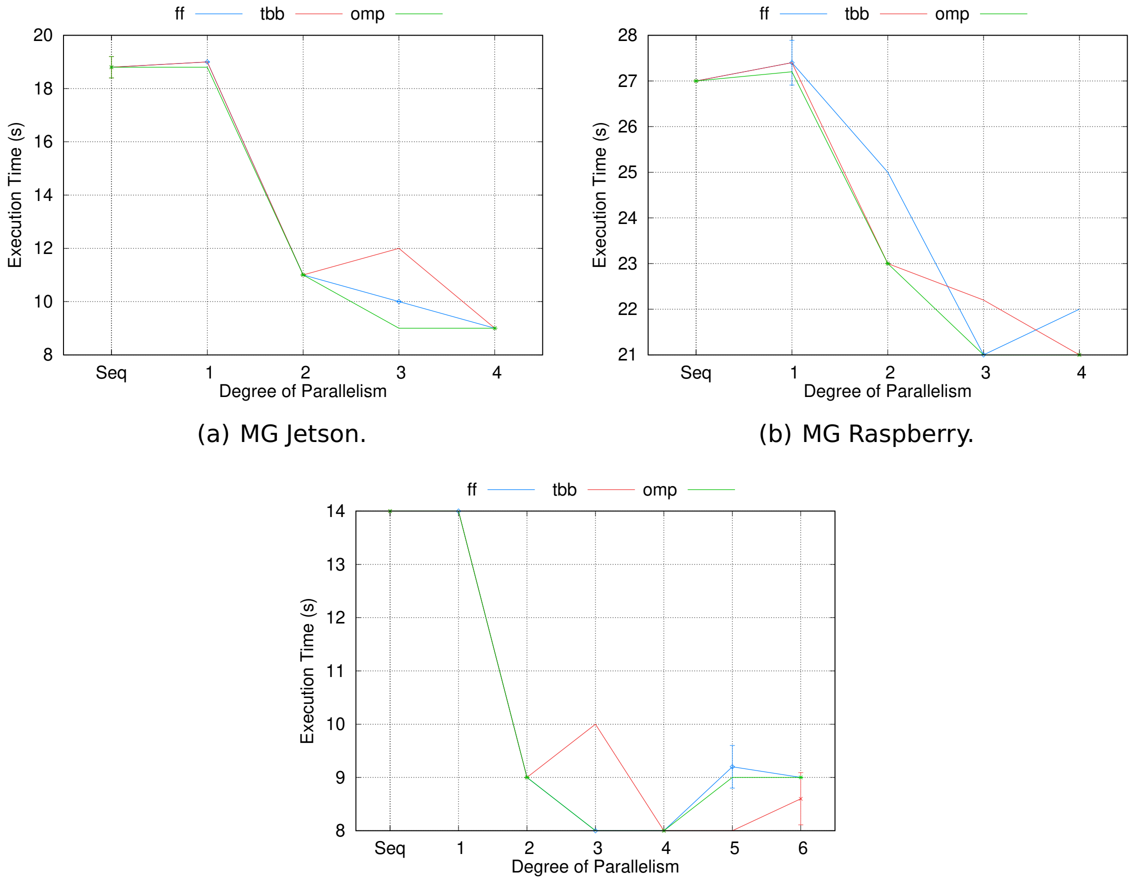
(c) FT Odroid.

Figure 5.5 – FT on multiple devices.

irregular data access in memory, which prevents the best cache use. EP, which has linear data access patterns, has 3.22% L1 cache load misses, while MG has 7.57%, and CG displays 33.81% L1 cache load misses. Higher L1 cache miss benefits hardware with higher L2 and L3 levels of cache sizes as they are more likely to contain, depending on the data access patterns, the required data for each computation instead of going to the main memory. However, our devices all have the same L1 cache hierarchy size and showcase similar scalability.

We observe the results of MG and CG on the graphs in Figures 5.6 and 5.7, respectively. The first characteristic we observe is that both have limited scalability. Since these applications have irregular data access patterns, the CPUs must wait considerably longer for data to arrive from the main memory instead of the L1 cache levels. Since MG has a lower cache miss rate, it can scale slightly better than CG; MG scales up to the degree of parallelism three, and CG mainly scales up to degree 2 (see Figures 5.6 and 5.7).

Comparing PPIs, we can also reinforce the RO3.6 on Odroid devices with degrees of parallelism 5 and 6. Another factor is that CG requires many MapReduce synchronizations, during which time the computation is sequential. Besides the fact that it limits



(a) MG Jetson.

(b) MG Raspberry.

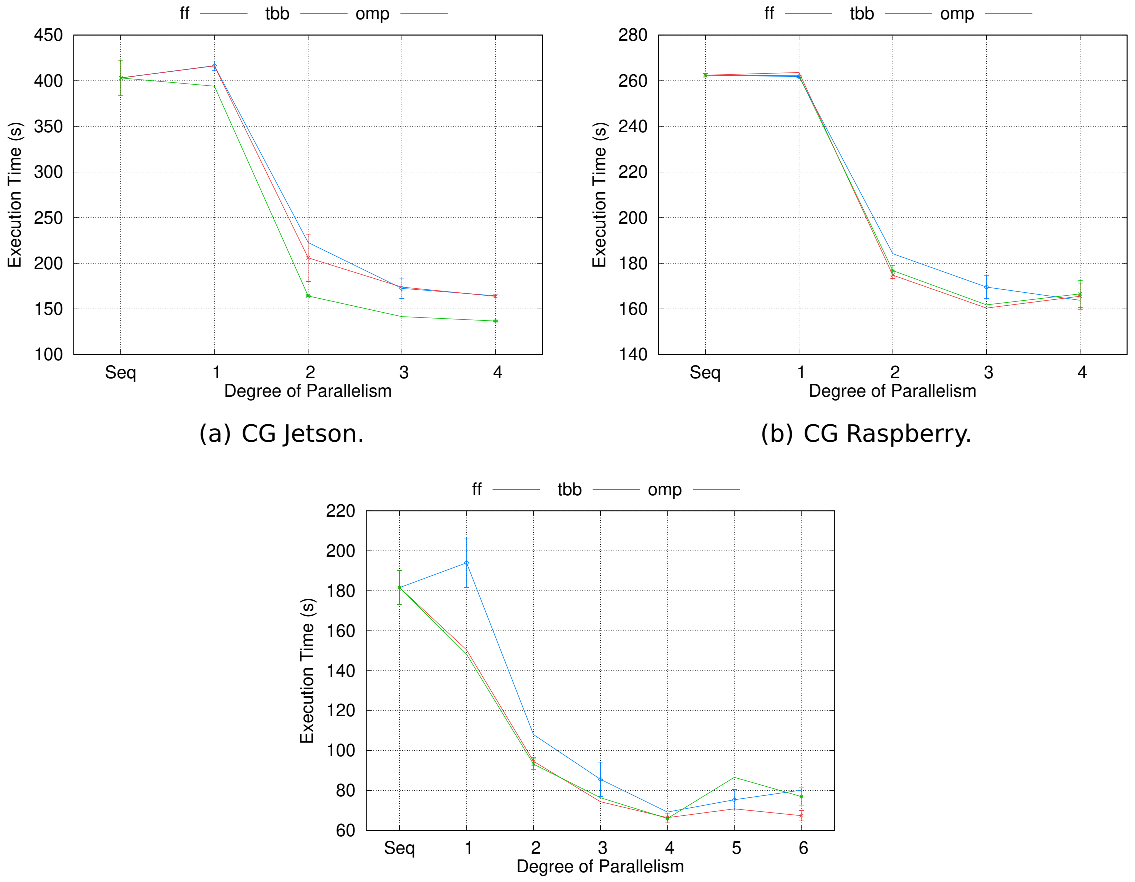
(c) MG Odroid.

Figure 5.6 – MG on multiple devices.

scalability, it also affects computation. Since it introduces implicit barriers, OpenMP is the only interface that supports adding no wait directives to remove them and continue computing the next round of results. TBB's work-stealing scheduler is more efficient than FF's static round-robin assignment at handling these applications' irregular workloads.

5.3.4 RO3.4: Evaluating Map with data dependency

LU implements the Symmetric Successive Over-Relaxation (SSOR) method to solve a linear system of equations. Its intensive computation relies on decomposing a 3D matrix system in triangular lower/upper matrices and then solving this matrix system. In this system, a thread can only start its computation after its neighbor has finished. The traditional data parallelism method is better because all three dimensions have data dependencies [9]. This alternative would require sending many update messages between parallel threads, impairing parallel scalability. Therefore, the alternative is using lock synchronization mechanisms. Parallelism happens when for each advance along one dimension, a new thread starts computing the next block of elements, which are incremental



(a) CG Jetson.

(b) CG Raspberry.

(c) CG Odroid.

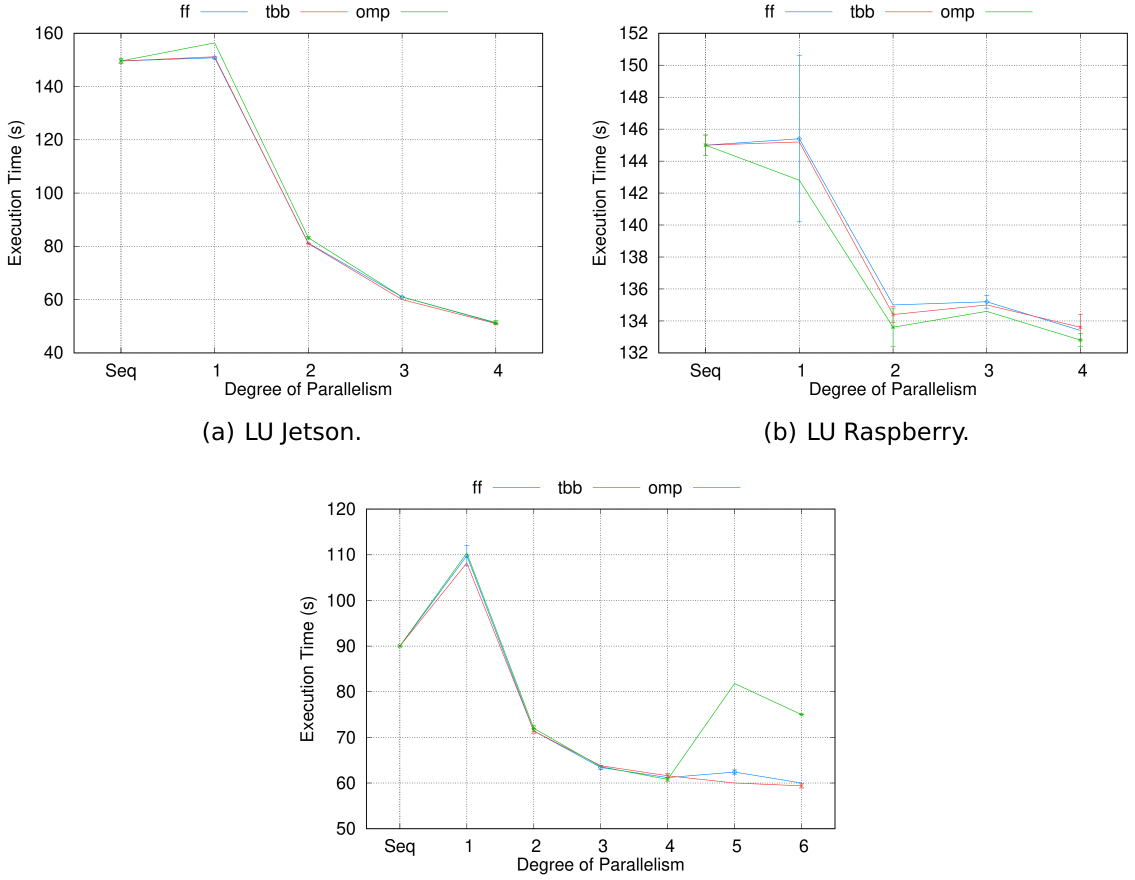
Figure 5.7 – CG on multiple devices.

higher. Effectively, LU is an application with high data dependency (across all three dimensions) that uses multi-step lock-unlock thread computations.

Figure 5.8 showcases the results of the LU application. On Raspberry, all application versions have small scalability with speedups no greater than 1.1. The main reason is heat dissipation issues, which we discuss in RO3.7. On the other hand, Jetson achieves 2.9 speedups and Odroid 1.59. On Odroid, OpenMP performs worse on degrees 5 and 6 for the same reasons we discuss on RO3.6. Despite that, we observe no particular advantage over any PPI over others. Ultimately, parallelism can be advantageous even with high data dependencies.

5.3.5 RO3.5: Analyzing memory bandwidth as a parallelism bottleneck

BT and SP implement implicit iterative methods to approximate the solution in CFD equations using a series of Maps without reduces or critical Sections contained. Between each Map pair, there is an implicit barrier logic. Regarding parallelism, one tech-



(a) LU Jetson.

(b) LU Raspberry.

(c) LU Odroid.

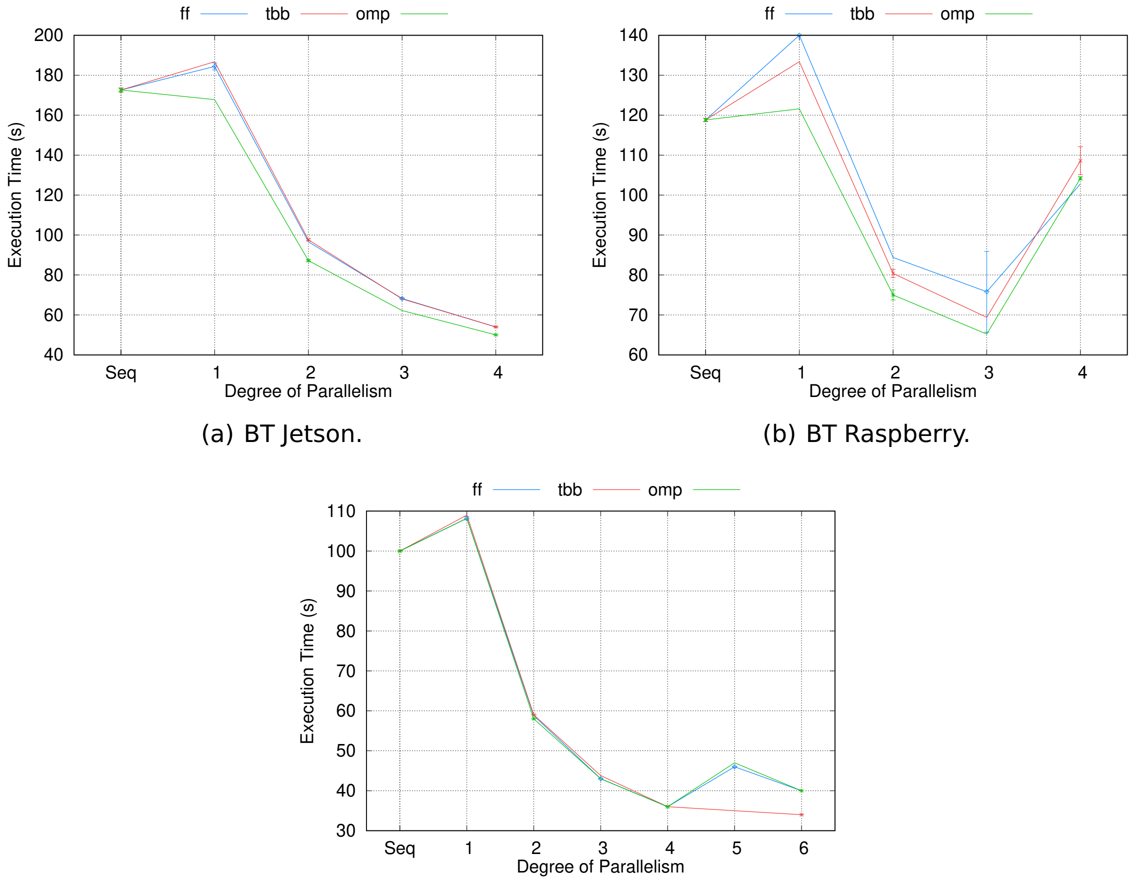
Figure 5.8 – LU on multiple devices.

nique to improve performance is re-utilizing the Map structures (6 within the main computation) to avoid repeatedly creating and destroying threads and organizing communication and synchronization structures. Furthermore, the application's scalability is limited by a sequential PDE (Partial Differential Equation) solver [54]. However, there needs to be more inherent parallelism in our devices to create this bottleneck.

Regarding the BT application, we observe good scalability in the graphs in Figure 5.9. Raspberry achieves a maximum 1.82 speedup, Jetson 3.45, and Odroid 2.77. On the other hand, the SP application has poor scalability in Figure 5.10. These are intriguing results because SP only has good scalability on the Jetson board (2.3 speedups). Raspberry (1.086 speedups) and Odroid (1.25 speedups) do not scale much and even deteriorate performance with higher degrees of parallelism. SP has a higher cache-miss rate at 5.82%, while BT is around 1.94%, and SP has 86% more branch misses than BT. We also analyze CPU utilization; both applications can maintain CPU utilization above 95%.

Using a Single Board Computer profiler [1], we obtain the memory bandwidth of Jetson is 3728.0 MB/s, Raspberry is 2478.0 MB/s, and Odroid is 3884.0 MB/s. SP is an-

[1] https://github.com/ThomasKaiser/sbc-bench



(a) BT Jetson.

(b) BT Raspberry.

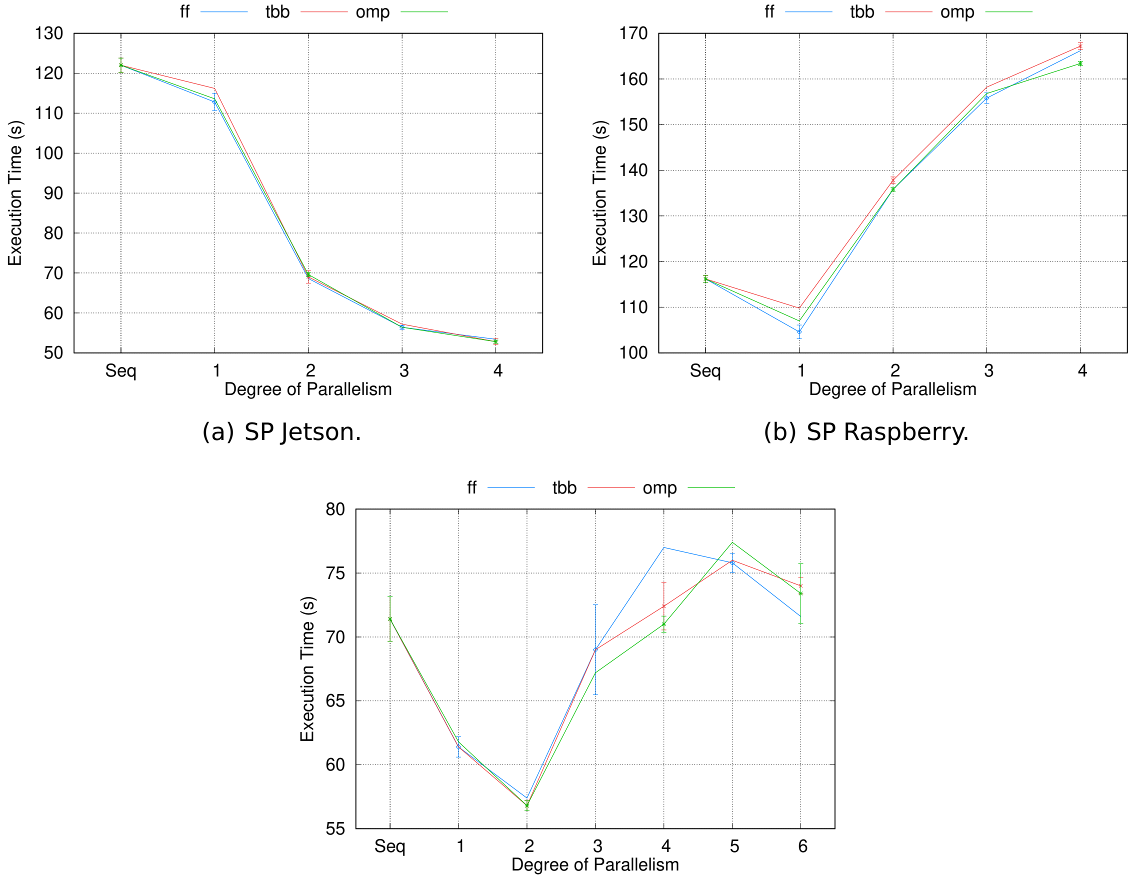
(c) BT Odroid.

Figure 5.9 – BT on multiple devices.

other memory-bound application, so the bandwidth explains why Odroid scales better than Raspberry. However, another difference still explains why Jetson is the only device that scales with parallelism. The most prevalent hardware difference between these three devices is the L2 cache size. Jetson has double the L2 cache size (2048 MB) as Raspberry and Odroid (1024 MB). In SP, threads process bigger chunks of data. Therefore, they also have more L1 cache misses (5.82% vs. 1.94%). Therefore, SP scales better on Jetson because it has more L2 cache. Given that Jetson has 2.3 speedups and Odroid has 1.25 speedups, double L2 cache size displays doubles the speedup. The speedup is also affected by available memory bandwidth.

5.3.6 RO3.6: Evaluating BIG.little architecture performance features

One important feature to consider is mapping threads to physical cores. The Odroid is a BIG.little architecture example with four high-performance and two low-performance processing cores. Some parallel interfaces may pin threads to physical cores to exploit cache locality better. Default `FastFlow` adopts this pinning strategy, although this be-



(a) SP Jetson.

(b) SP Raspberry.

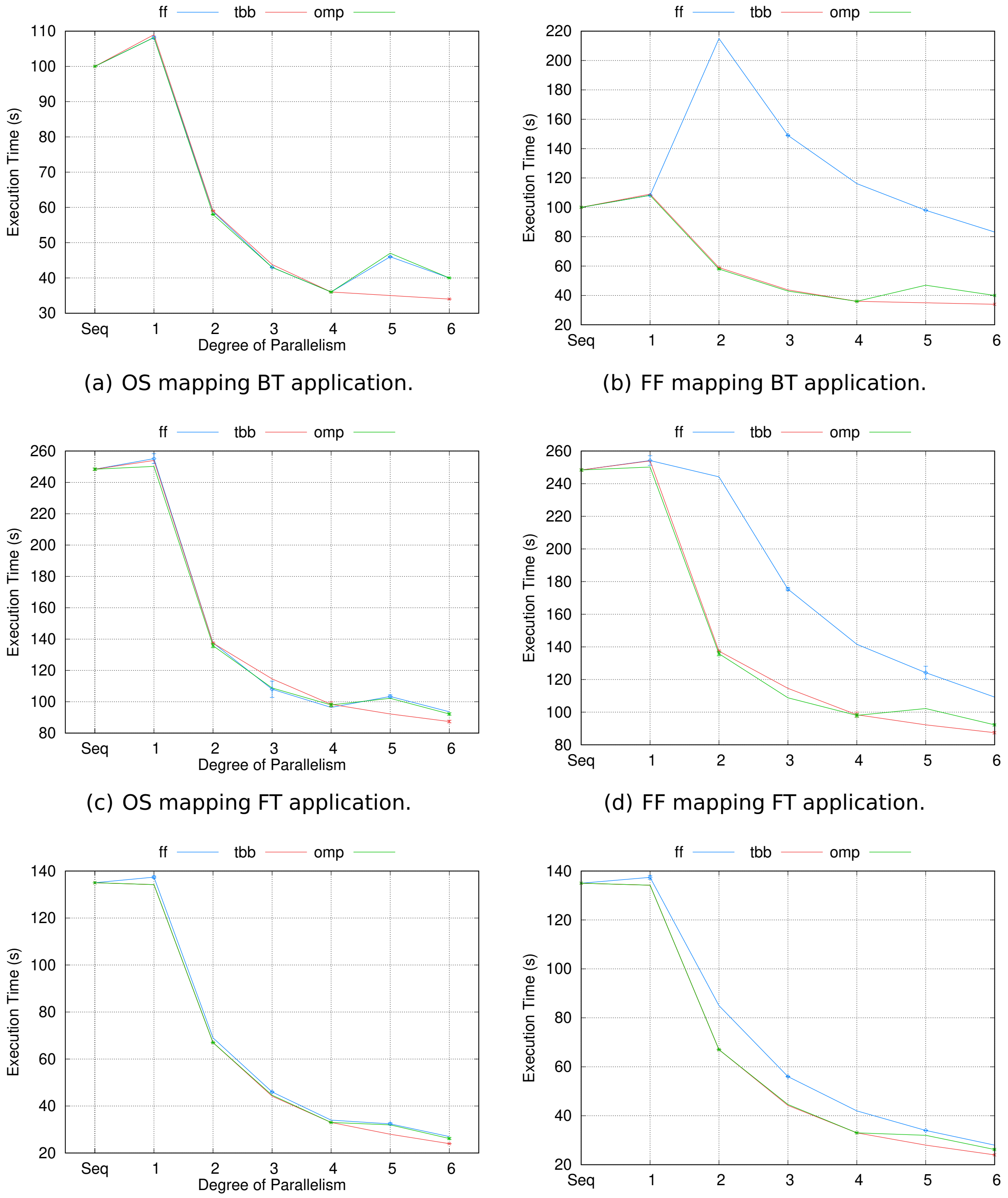
(c) SP Odroid.

Figure 5.10 – SP on multiple devices.

havior is customizable. Some threads are pinned to low-performance cores. It is known that the slowest part of the parallel program is the limiting factor; in this case, it is the threads that execute on the low-performance cores. If well used, low-performance cores can increase performance; however, efficient parallel programs must also consider this architectural feature. Typically, the OS of Odroid can handle this sufficiently well. We can observe this phenomenon in Figure 5.11, where the graphs on the left showcase BIG.little aware OS scheduling, and the graphs on the right showcase the non-BIG.little aware of FF mapping. In this case, the OS BIG.little aware scheduling can improve performance by up to 366%. This is a similar conclusion to previous research [18].

5.3.7 RO3.7: Analysis of the impact of hardware temperature on performance

The consumption metrics we evaluate in RO4 show that using the maximum available hardware resources (via parallel computing) increases power dissipation. Even though the operating frequency is the same (they all use performance mode operating system governor), parallelism means more CPU nodes are operating, internal bus data



(a) OS mapping BT application.

(b) FF mapping BT application.

(c) OS mapping FT application.

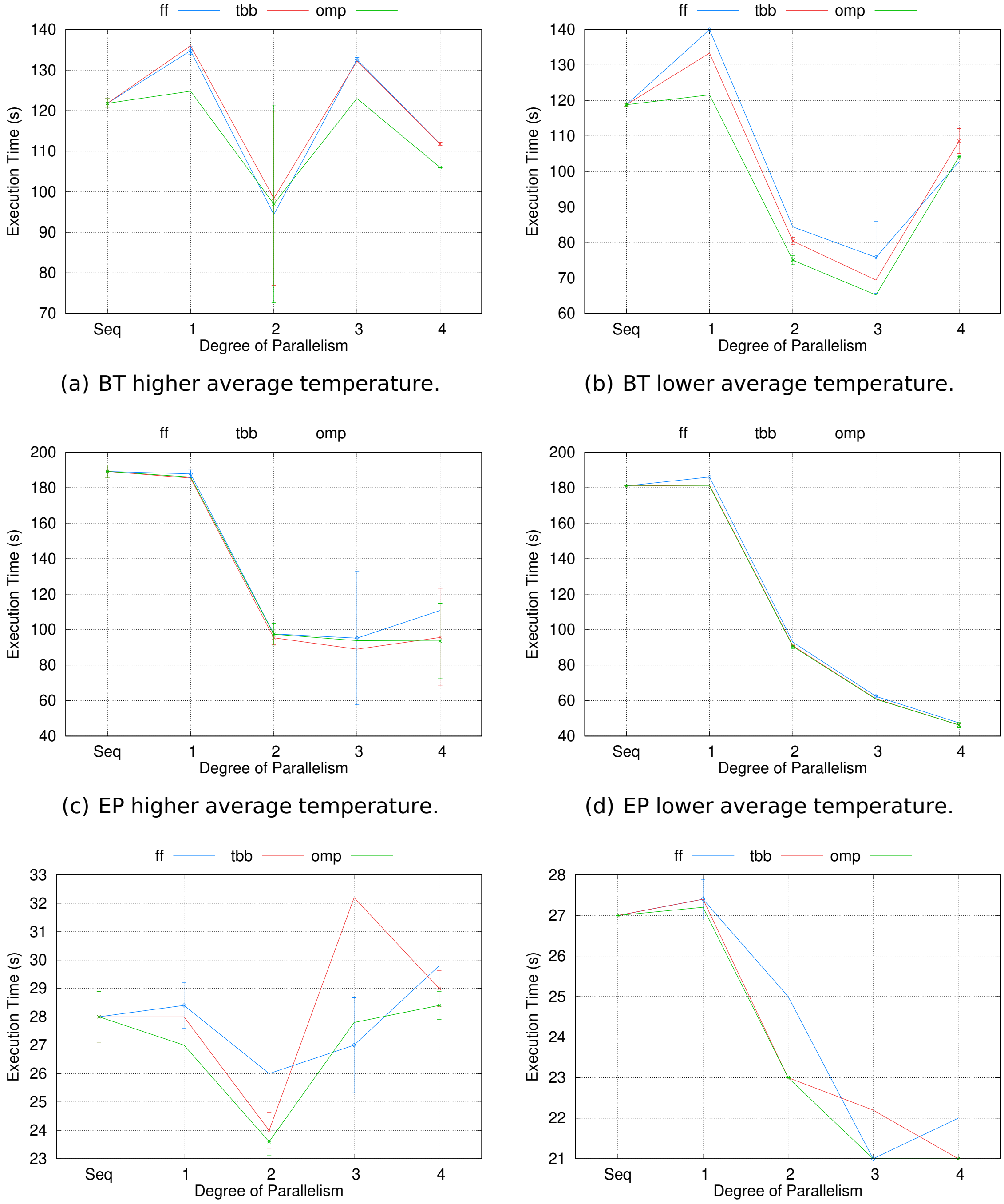
(d) FF mapping FT application.

(e) OS mapping EP application.

(f) FF mapping EP application.

Figure 5.11 – Odroid proof for a correct thread-to-core mapping.

transfers, and increased memory usage across all hierarchical levels. Therefore, higher power dissipation means the hardware dissipates more heat than the lower power. High temperatures in processors can lower performance. According to the documentation, the CPU temperature range for Raspberry is up to 85ºC. Our experiments showed that our Raspberry device setup could not dissipate heat as efficiently as Odroid and Jetson. The problem happens during prolonged periods of maximum parallelism when temperatures reach 75ºC. The idle temperature is around 51.6ºC in a 22ºC room.



(a) BT higher average temperature.

(b) BT lower average temperature.

(c) EP higher average temperature.

(d) EP lower average temperature.

(e) MG higher average temperature.

(f) MG lower average temperature.

Figure 5.12 – Raspberry device.

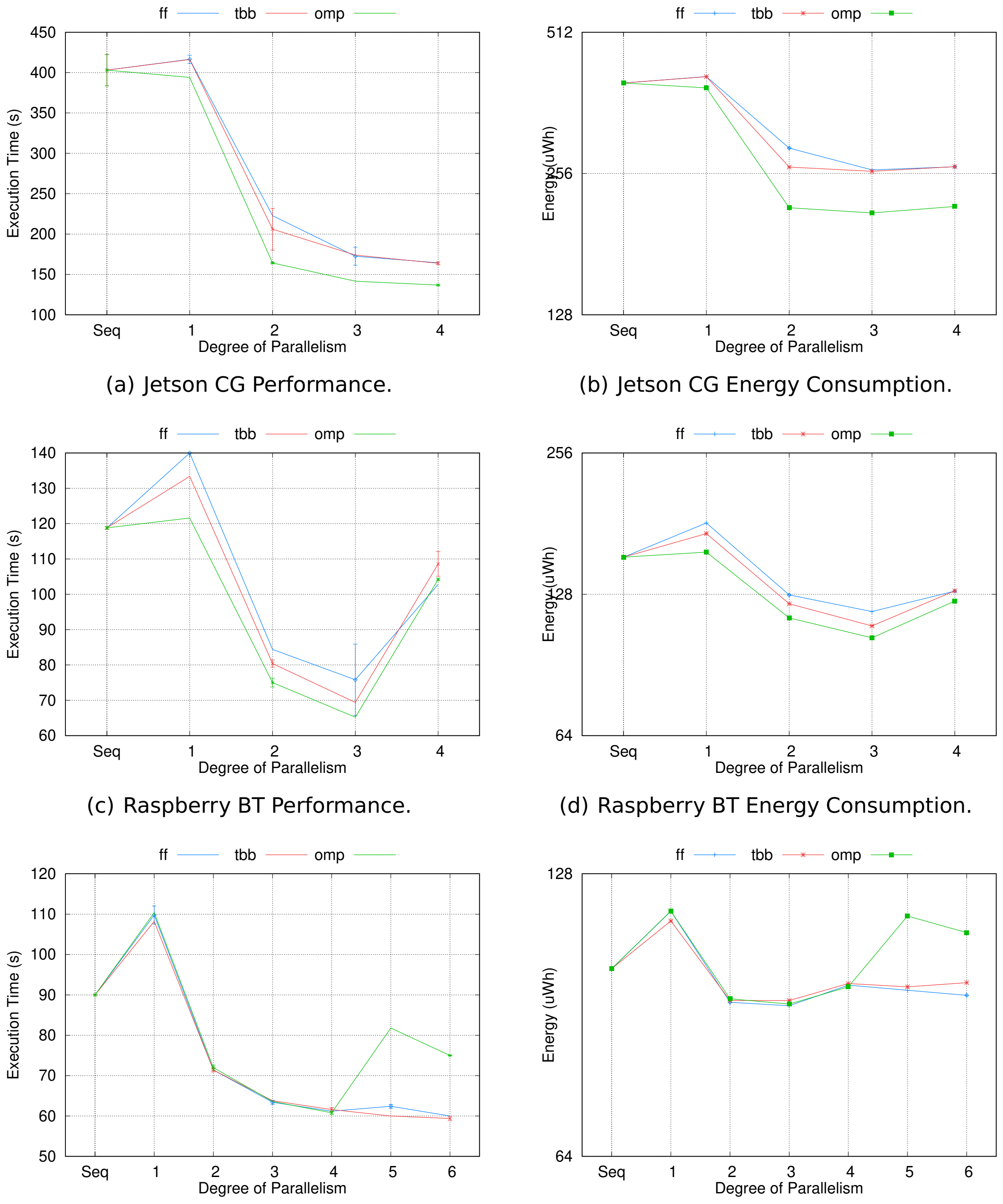
To confirm that temperature is the issue, we perform some extra experiments. After finishing an experiment using maximum parallelism and achieving temperatures as high as 75ºC, we let the device cool down to its idle temperature, which usually takes 30 to 60 seconds in our setup. We start the next experiment only after the temperature is back in its idle state. By keeping the average temperature of the device lower, we can improve the total (all 8 applications) average performance from 99.425 seconds to 83.52 seconds, representing a 17.39% better performance considering all parallel versions. However, if we look at the individual applications, we see performance improvements ranging from smaller 0.9% (SP) up to 49.89% (EP).

To help visualize the results, we showcase in the graphs in Figure 5.12 the higher-temperature performance readings on the left and the lower-temperature readings on the right. Comparing the graphs on the right and the left, we see that the lower degrees of parallelism (Seq and 1) do not improve execution time as much as higher degrees of parallelism (3 and 4). The reason is that, as discussed in the previous paragraph, the average temperature is not as high when the parallel application uses fewer CPUs. We observe a maximum temperature of 66ºC with a degree of parallelism compared to 75ºC with degree 4. We can perform better (0.9% up to 49.89%) by keeping the processor with a lower average temperature.

5.4 RO4: Evaluating consumption metrics for data processing applications

Given that the **Context** is equivalent to RO5, regarding **Forces**, the PPIs that use the most hardware resources yield the most power dissipation but the lowest energy consumption. They use the hardware resources most intensively, momentously drawing more power, which results in higher computational power. For instance, a PPI that uses all CPUs simultaneously will necessarily draw more power. However, PPIs that potentially cause less cache-miss also cause fewer data movements within the hardware and may reduce energy consumption because fewer actions should occur. It also reduces power dissipation. Parallelism can cause more hardware operations executed (i.e., synchronization, communication, and more L1 cache movements due to more cores). Conversely, lower execution time from parallelism lowers the total energy consumption by lowering the time interval that measures the idle component's energy consumption. Other hardware and system considerations also affect energy consumption (i.e., clock frequency and peripherals activity); however, they are not related to PPIs parallelism. The main **limitations** of this analysis is related to the precision of the energy measurement device and sampling rate of 3 times per second.

Discussion - Often, energy consumption has some degree of correlation with their performance counterparts; when the execution time is lower, the total energy consumption is also typically lower because the processing finishes earlier, and so does the energy consumption reading. This effect indicates that applications that take longer to execute consume more energy. The constant energy consumption by idle components (i.e., GPUs, peripherals, video output, and DMA) also increases this effect. However, this is only sometimes the case. We highlight in Figure 5.13 the performance graphs on the left and energy consumption graphs on the right. These are a few examples of some applications among multiple devices in which we can see a correlation between performance and energy consumption; energy consumption and performance might change at different rates. If we apply a Pearson correlation calculation between these two values, we get a correlation factor of 0.51.



(a) Jetson CG Performance.

(b) Jetson CG Energy Consumption.

(c) Raspberry BT Performance.

(d) Raspberry BT Energy Consumption.

(e) Odroid LU Performance.

(f) Odroid LU Energy Consumption.

Figure 5.13 – correlation between execution time and energy consumption on data processing applications.

Table 5.6 summarizes the energy consumption in uWh for all NPB applications across all devices. We can highlight that, according to the total average values, parallelism can reduce the total energy consumption required to perform the same set of computations by 30.36 to 31%. In contrast, the difference in execution time improvements regarding the sequential version is between 56.57 and 57.96%. Therefore, the gains in energy consumption are smaller than performance but still significant.

Comparing devices, the energy Odroid consumes, on average, 32.95% and 57.97% less energy than Jetson and Raspberry, respectively. In comparison, Odroid has, on average, 46.94% and 54.65% better performance than Jetson and Raspberry, respectively. In this case, the trend repeats where gains in energy consumption are smaller than performance. Overall, Odroid achieves less execution time and more energy consumption because it has more inherent parallelism (6 cores instead of Raspberry and Jetson 4).

Table 5.6 – Energy consumption for NPB. The maximum observed standard deviation is 0.034. For parallel versions, green and red highlight the best and worst observed results respectively.

Device	**Application**	**Total Energy Consumption (uWh)**			
		FastFlow	**OpenMP**	**TBB**	**Seq**
Jetson	BT	94.60	89.00	95.60	164.20
	CG	264.60	217.80	264.80	399.00
	EP	116.80	113.80	113.80	281.00
	FT	137.80	138.40	138.00	266.20
	IS	8.80	8.40	7.80	18.60
	LU	85.20	85.80	86.40	144.00
	MG	21.00	18.80	20.00	24.40
	SP	90.20	91.40	90.80	122.00
Raspberry	BT	130.00	123.80	130.20	153.60
	CG	264.80	265.80	264.80	346.80
	EP	74.40	72.20	72.40	197.00
	FT	201.60	200.00	199.40	394.00
	IS	8.00	7.80	8.20	16.80
	LU	201.20	200.20	201.20	182.80
	MG	40.80	38.80	39.20	39.40
	SP	250.40	248.00	252.60	148.20
Odroid	BT	73.80	74.00	72.40	106.60
	CG	140.00	137.40	133.40	206.40
	EP	39.00	38.20	36.00	108.00
	FT	137.60	137.20	136.60	233.00
	IS	3.40	4.00	4.00	6.80
	LU	95.00	110.80	98.00	101.40
	MG	17.80	17.80	18.40	19.00
	SP	113.40	117.40	120.60	81.00
Total Avg.		108.76	106.53	108.53	156.68

Examining Table 5.6, we see that OpenMP yields the best overall energy consumption readings. FastFlow and TBB perform similarly to each other (difference of 0.21%) but are, respectively, 2.09% and 1.84% inferior to OpenMP. Overall, the best-performing interfaces in each device and application yield the lowest energy consumption. The explanations for all readings are the same as for equivalent performance results in RO3. However, the evaluation can also take into consideration power dissipation.

Table 5.7 summarizes the average power dissipation in W for all NPB applications across all devices. Therefore, as expected, since the sequential version uses the least amount of hardware resources, it consumes the least average power. Considering that the reduction in energy consumption from parallel and sequential versions is 30.36 to 31%, the increase in power dissipation between parallel and sequential versions is between 65.77 and 68.05%.

Another vital factor to consider is power peaks. Isolating the maximum Power reading during the execution of the application across all devices and calculating the average, we get that `FastFlow` peaks at 6.36 W, `OpenMP` 6.42 W, TBB 6.61 W, and the sequential at 4.68 W. Additionally, if we observe absolute single maximum values, we get `FastFlow` with 7.982 W, `OpenMP` with 8.123 W, TBB with 8.338 W, and the sequential with 6.468 W. Therefore, the average and maximum peaks, we get that `FastFlow` is the version that achieves the least power peaks, followed by `OpenMP`, and then TBB. However, this represents at most 4.48% of the difference between them. In this case, we only compare a single device; this peak difference becomes more relevant in a cluster with hundreds or thousands.

Overall, this leads us to conclude that `OpenMP` is the most suitable interface for data processing applications. `FastFlow` power peaks are 1.76% lesser than OpenMP; however, `FastFlow` consumes total energy. `TBB` and `FastFlow` are very similar to each other regarding energy consumption.

5.5 RO5: Evaluating performance metrics of stream processing applications

Context - Parallel stream processing allows multiple threads to process data simultaneously whenever available. Typical metrics to measure the performance of stream processing systems are latency, throughput, and resource utilization. Resource usage is considered in RO6. Latency is the average time between a stream item generation and the moment it finishes processing. This metric is essential to systems that operate under some degree of real-time constraints. Throughput measures the average processing rate of items per second; it indicates that the system can process more items in a period of time.

Forces - Past research shows that the relation between these metrics displays some exclusivity [71]. The parallel algorithm can allocate the maximum number of processing resources to one of the stream items as soon as it arrives. Alternatively, it can opt for a more balanced approach that distributes the processing resources among different stages. The first option can reduce latency to the detriment of throughput, while

Table 5.7 – Power dissipation for NPB. The maximum observed standard deviation is 0.61. For parallel versions, green highlights the best and red the worst results.

Device	Application	Average Power Dissipation (W)			
		FastFlow	OpenMP	TBB	Seq
Jetson	BT	6.17	6.30	6.24	3.38
	CG	5.37	5.25	5.39	3.42
	EP	4.33	4.34	4.35	2.69
	FT	5.49	5.61	5.57	3.13
	IS	4.77	4.95	4.86	2.87
	LU	5.84	5.87	5.92	3.40
	MG	5.42	5.18	5.28	3.73
	SP	5.97	6.06	6.05	3.57
Raspberry	BT	4.53	4.21	4.27	4.60
	CG	5.43	5.37	5.37	4.52
	EP	5.56	5.54	5.54	3.99
	FT	5.34	5.31	5.32	4.17
	IS	5.20	5.20	5.40	4.03
	LU	5.35	5.37	5.34	4.47
	MG	5.42	5.39	5.42	4.38
	SP	5.36	5.40	5.38	4.53
Odroid	BT	6.57	6.54	7.41	3.77
	CG	5.68	5.80	6.31	3.88
	EP	5.10	5.09	5.37	2.86
	FT	4.98	5.05	5.29	3.17
	IS	5.21	5.54	5.16	3.13
	LU	5.61	5.22	5.84	4.02
	MG	5.60	5.51	5.83	4.05
	SP	5.60	5.67	5.78	4.00
Total Avg.		5.41	5.41	5.53	3.74

the second can potentially increase throughput to the detriment of latency. Another parallel algorithm feature is considering statically or dynamically allocating resources for the computation.

The main **threats to the validity** of this RO and its derived ROs are related to the limited variations in characteristics of the parallelism algorithms in SPBench applications. All applications, except Ferret, have the same parallel pipeline structure. Another point to consider is related to latency measurement. We attach a timestamp to the stream items the moment those items are generated in the first processing stage of the pipeline. This benefits TBB because it only executes this stage and, therefore, the timestamp creation when it has available resources. Therefore, TBB’s latency is potentially lower than if an independent external system generated the stream items.

Discussion - Table 5.8 showcases latency measurements for all SPBench applications using the maximum hardware resources. The first evident conclusion is that using parallelism increases latency. The reason is that, in addition to the original computation, parallelism adds to the cost of communication, data movements, and synchronization of the computation. Comparing our experiments' average latency, the best parallel version (TBB) has 37% higher latency. TBB's latency measurement difference is the resource allocation strategy, which we examine later in this Section. Threads and OpenMP increase latency by 230.48% – they use the same queue to communicate between stages – while FastFlow increases by 264.63%. This difference between Threads/OpenMP and FastFlow is due to the data communication queue's implementation strategy. FastFlow adopts multiple lock-free queues between each producer-consumer thread pair, while Threads/OpenMP uses a single lock-guarded queue between each stage. When the resource contention is higher (using a higher degree of parallelism), the Threads/OpenMP approach yields a more balanced workload distribution. We can visualize this situation in the left-side graphs in Figure 5.15, where Threads/OpenMP consistently achieve lower latency with the maximum degree of parallelism.

On the other hand, SPar achieves the most significant average latency readings by a large margin. SPar typically performs similarly to FastFlow; Ferret is an exception where SPar achieves very high latency readings and inflates the total average (see Figure 5.15 (b)). Even the standard deviation for SPar Ferret is relatively high: 0.45 seconds. The reason is that SPar generates an inefficient code for latency in this application. SPar generates a pipeline of farms (i.e., $pipe(farm(stage_1), ..., farm(stage_n))$, and the FastFlow implementation generates an optimized pattern composition (i.e., $pipe(pipe(farm(stage_1))$, ...,$pipe(farm(stage_n)))$. The SPar version for Ferret generates four empty data emitters and collectors that, although they do not significantly degrade throughput, add extra steps to the communication path and severely degrade latency. These four communication paths add extra queues with a 512 default size that severely degrades latency. The graph in Figure 5.14 exemplifies this situation, where the on-demand paths have a queue size of 1, and the four instances where it does not have on-demand have 512 queue sizes. The user does not control this as it is a SPar internal. This observation points to the fact that shorter data paths reduce latency. Furthermore, we can observe that on-demand scheduling (queue-size 1) or dynamic scheduling can greatly reduce latency.

Table 5.9 showcases the throughput results for all SPBench applications across all three devices, including the total average at the bottom. In this case, the individual application results mostly correspond to the total average; TBB has the highest throughput, Threads, and OpenMP closely match TBB. Then SPar and Fastflow – similar to each other – come last. Approximately, SPar/FastFlow achieves a speedup of 3.15, and the other three PPIs achieve 3.51 using the maximum available resources. These results confirm that par-

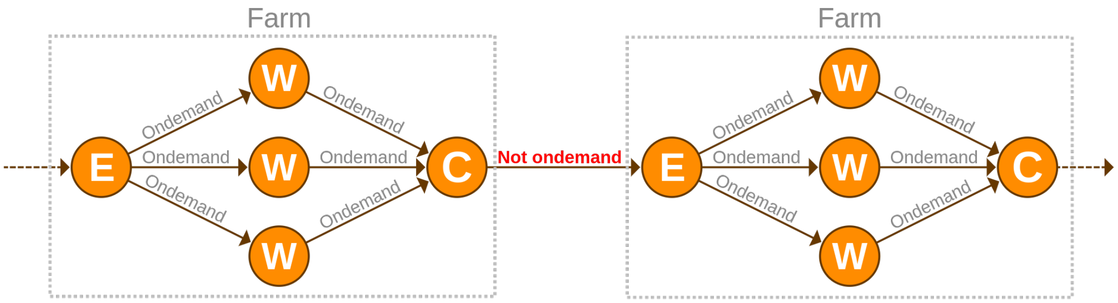


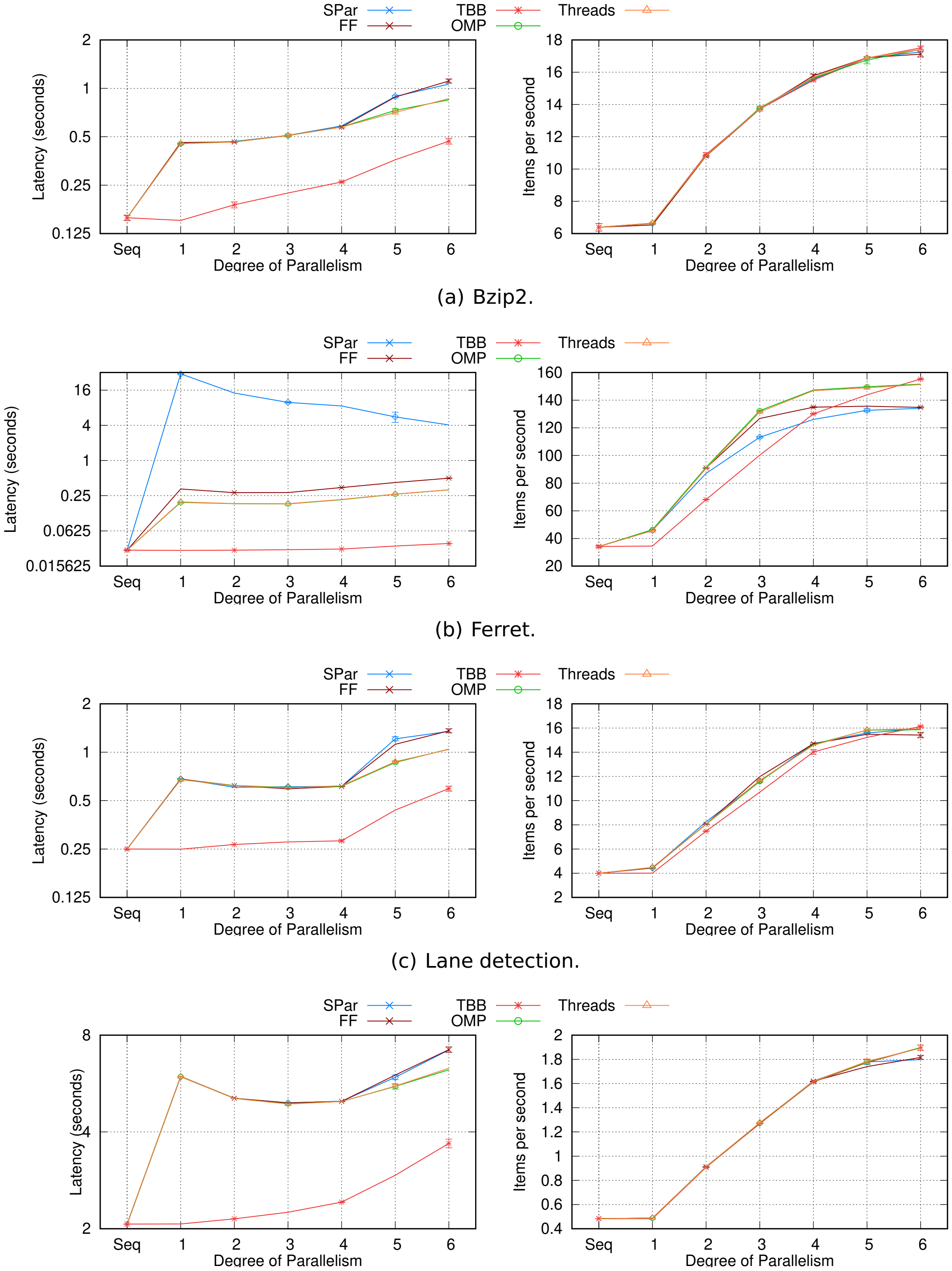
Figure 5.14 – Example of the problem in SPar code generation.

Table 5.8 – Latency measurements for all SPBench applications. The standard deviation can vary between 0.0003 and 0.45. For parallel versions, green and red highlight the best and worst observed results respectively.

Device	Application	Latency (seconds)					
		Seq	SPar	Fastflow	TBB	Threads	OpenMP
Jetson	Bzip2	0.24	0.73	0.73	0.29	0.72	0.71
	Ferret	0.05	49.22	0.74	0.05	0.44	0.44
	Lane	0.93	2.65	2.65	0.97	2.66	2.68
	Face	3.40	9.82	9.79	3.81	9.75	9.74
Raspberry	Bzip2	0.21	1.19	1.20	0.45	1.01	1.01
	Ferret	0.04	40.57	0.88	0.06	0.50	0.51
	Lane	0.33	1.24	1.26	0.50	1.13	1.17
	Face	2.10	8.67	8.48	3.76	7.89	7.86
Odroid	Bzip2	0.16	1.06	1.11	0.47	0.86	0.85
	Ferret	0.03	30.36	0.50	0.04	0.32	0.31
	Lane	0.25	1.35	1.36	0.59	1.04	1.05
	Face	2.07	7.19	7.21	3.68	6.32	6.25
Total Avg.		0.82	12.84	2.99	1.22	2.72	2.71

allelism can significantly increase the performance of stream processing applications on limited-resource hardware. The graphs also showcase this on the right side of Figure 5.15.

The graphs in Figure 5.15 show that Latency and Throughput do not directly correlate. For instance, we observe that in Bzip2, we have a range difference ($(max - min)/average$) of 68.99% and 6.33% for latency and throughput, respectively. However, Ferret has a range difference of 490.27% and 96.02% for latency and throughput, respectively. Additionally, latency and throughput are not exclusive (inversely correlated). As we can observe in all degrees of parallelism 2, higher latency does not entail lower throughput and vice-versa. For instance, with a degree of parallelism 2, we get 659.9 milliseconds latency and 7.54 items/second throughput; with a degree of parallelism 4, we get 692.69 milliseconds latency and 13.08 items/second throughput. Even though the throughput increases by 73.54%, the latency only increases by 4.97%. The Pearson correlation factor between throughput and latency is 0.197, which indicates a low correlation.



(a) Bzip2.

(b) Ferret.

(c) Lane detection.

(d) Face Recognition.

Figure 5.15 – Odroid device graphs.

Table 5.9 – throughput of all SPBench applications. The maximum observed standard deviation is 0.92. For parallel versions, green and red highlight the best and worst observed results respectively.

Device	Application	Throughput (items/second)					
		Seq	SPar	Fastflow	TBB	Threads	OpenMP
Jetson	Bzip2	4.21	13.09	13.07	13.94	13.80	13.80
	Ferret	18.57	64.58	64.32	73.39	73.54	73.11
	Lane	1.07	4.02	4.02	4.09	4.14	4.10
	Face	0.29	0.99	0.98	1.03	1.03	1.02
Raspberry	Bzip2	4.70	8.23	8.35	8.97	8.91	8.96
	Ferret	25.34	55.28	55.77	65.94	64.40	64.06
	Lane	3.03	7.64	7.69	7.95	8.03	7.76
	Face	0.48	1.02	1.02	1.04	1.03	1.04
Odroid	Bzip2	6.39	17.26	17.13	17.51	17.42	17.42
	Ferret	34.15	134.11	135.66	155.21	151.30	151.64
	Lane	4.00	15.94	15.48	16.13	15.95	15.86
	Face	0.48	1.80	1.82	1.90	1.89	1.90
Total Avg.		8.56	27.00	27.11	30.59	30.12	30.06

5.5.1 RO5.1: Assessing the effect of blocking vs. non-Blocking computation on performance metrics

As discussed in RO1, SPar, and Fastlow permit selecting blocking or non-blocking behavior for waiting. Blocking may also be referred to as busy-waiting. Blocking behavior can yield better performance because threads waiting with NOP operations can instantly retrieve and compute the stream items. At the same time, non-blocking threads need to wait for the operating system to permit them to use the CPU and retrieve the context to continue computing [22]. Therefore, blocking behavior can potentially increase performance metrics if the application does not have considerable periods of data scarcity. Threads and OpenMP use C++ Mutexes to guard the parallel queue, which implement a partial spin-lock. The Mutex with a partial spin-lock emulates the blocking behavior for a fixed amount of time, after which it yields the processor back to the OS.

To assess this, we test stream processing applications on the exact condition using FastFlow blocking and non-blocking configuration. Effectively, the non-blocking approach yields 68.1% less throughput and increases the end-to-end latency by 205.9%. For low-contention processing stages, meaning that it typically does not have to wait for data to be available in the queue, this increase is less relevant at 32.6%. In conclusion, blocking communication between parallel stages can significantly improve performance metrics.

5.5.2 RO5.2: Evaluating computational resource allocation strategies in parallelism techniques

One important parallelism design choice is mapping computation to existing threads; it can be a static or dynamic approach. One thread performs only one logical type of computation using the static strategy. On the other hand, the dynamic approach permits one thread to perform multiple types of computation according to some scheduler logic. For instance, suppose we have computation $f(g(h(C)))$ where $C = x, y, z$ that applies filter h then g and then f to a data set containing x, y, and z. For the static approach, one thread always executes the same filter computation. In our example, one thread performs $f(C)$, another $g(C)$, and other $h(C)$. For the dynamic approach, one thread can perform any of the three filters computation according to some scheduler logic. In our example, one thread performs $f(g(h(x)))$, another $f(g(h(y)))$, and other $f(g(h(z)))$. The dynamic approach could also have other valid combinations.

The main difference in performance between static and dynamic computation and thread mapping is the logical set of actions taken when resource contention occurs. In the dynamic approach, the scheduler logic must detect the bottleneck, look for other available computations, and then change the thread behavior accordingly. There is a computational overhead for the scheduler logic. However, when a static thread can not perform its computation, it must either yield the physical core through an operating system call or it can block using NOP (no operation) instructions. Here, the computational overhead occurs with the system calls and context switching or meaningless NOP computations. Out of our tested PPIs, only TBB uses the dynamic approach. Every other interface uses static.

TBB consistently achieves lower latency readings due to its thread allocation strategy. TBB threads can execute any pipeline computational filter or stage, which is coordinated by its dynamic task scheduler. Therefore, TBB threads only execute the stream generator code when it has sufficient computational resources, which acts as a pipeline back pressure mechanism. On the other hand, the static resource allocation strategies have one dedicated thread for the stream generation computation. This thread also computes dynamically, only generating items when the following stage can compute them. Overall, even with an extra logic task scheduler cost, the dynamic approach yields a better latency (122.95%) because it better exploits that extra thread's computational power. This is particularly relevant in the limited-resource environment. The graphs on the left of Figure 5.15 illustrate this behavior.

However, another effect that results from the static vs. dynamic resource allocation approach is perceptible from the right-side throughput graphs of Figure 5.15: TBB (dynamic) can yield lower throughput than the other PPIs (static) when using a lower degree of parallelism. This difference can reach 12.73% and 26.469% for Lane detection and Ferret, respectively. However, for Face recognition and Bzip2, this difference is primarily negligible at 0.043287%. It leads us to conclude that the dynamic approach can better handle resource contention – using the maximum number of available computational resources – improving by at least 1.72% up to 16.74%. On the other hand, the static approach achieves slightly better performance when there are exceeding available resources (ranging from 0.043287% up to 26.469%).

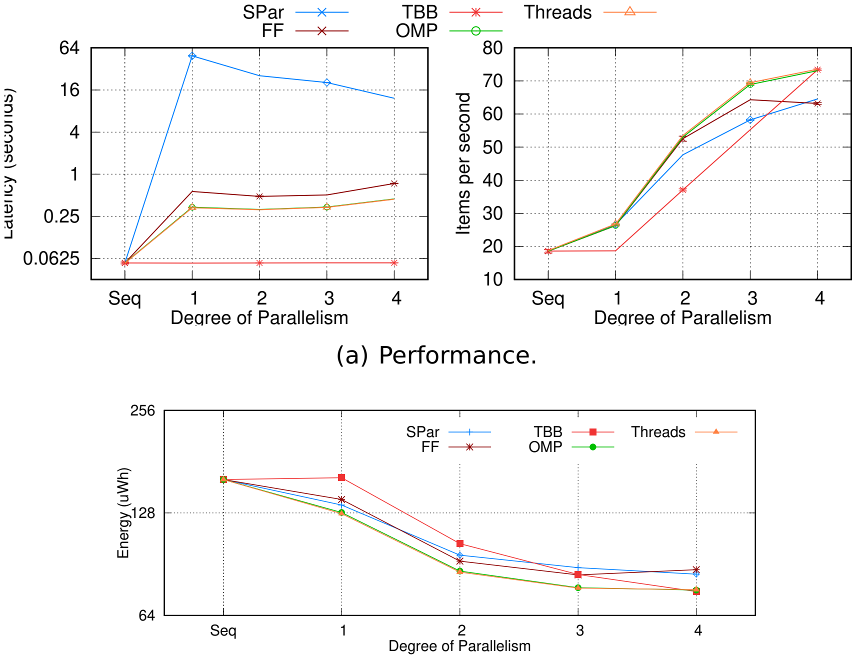
5.6 RO6: Evaluating consumption metrics for stream processing applications

The **Context -** for stream processing applications is similar to RO5. For **Forces -**, we also consider memory consumption to assess the amount of extra memory each PPI introduces despite the natural parallelism replication. There is the effect of dynamic vs. static memory allocating strategy. The **limitations** are the same as stated in RO4.

Discussion - As we highlight for the data processing applications, there is some correlation between energy consumption and performance metrics. Figure 5.16 demonstrates that whenever an application achieves lower throughput (graph on the top right corner), it also has a higher energy consumption rate. If we apply a direct correlation calculation between execution time (used to calculate throughput) and energy consumption, however, we get a small correlation factor of 0.25. Data processing applications have a correlation factor of 0.51. We highlight that this is just a mathematical representation of our data set and is not a universal correlation between execution time and energy consumption.

Table 5.10 showcases the total energy consumption for all SPBench applications. Similarly to data processing applications, parallelism reduces the total energy consumption required to perform the same computations by 54.29 to 63.53%. It is a significant step up the data processing application range from 30.36 to 31%. For stream applications, we see that the improvement range in throughput regarding the sequential version is 103.7 up to 112.54%. Like the data processing applications, the energy consumption gains are smaller than performance but still significant.

Comparing devices, the energy Odroid consumes, on average, is 50.59% and 59.74% less energy than Jetson and Raspberry, respectively. In comparison, Odroid has, on average, 68.57% and 78.71% better performance than Jetson and Raspberry, respectively. In this case, the trend repeats where gains in energy consumption are smaller than performance. Odroid achieves better performance and energy consumption because it has more inherent parallelism (6 cores instead of Raspberry and Jetson 4).



(a) Performance.

(b) Energy Consumption.

Figure 5.16 – Ferret application: correlation between performance and energy consumption.

Table 5.10 – Energy consumption of all SPBench applications. The maximum observed standard deviation is 0.00089. For parallel versions, green and red highlight the best and worst observed results respectively.

Device	Application	Total Energy Consumption (uWh)					
		Seq	SPar	Fastflow	TBB	Threads	OpenMP
Jetson	Bzip2	79.8	45	45.4	42.6	43	42.8
	Ferret	160.4	84.6	87.2	75.2	76.2	76
	Lane	64.8	28.4	28.2	27.8	27.6	27.2
	Face	93.4	50.2	50.2	48.8	48	48.2
Raspberry	Bzip2	99.6	70.6	71.8	66.4	66	66.6
	Ferret	166.6	97.4	100.8	79.2	80.6	81.8
	Lane	37.8	18	18.2	17.4	17.2	17.6
	Face	82.4	50	49.2	46.4	45.8	46
Odroid	Bzip2	54.8	35.8	36.2	35.8	34.6	35.6
	Ferret	88	47.4	47	41.2	42.4	41.8
	Lane	20.6	10.6	11.6	10.4	11	10.8
	Face	56	29.6	29.6	28.8	29	28.6
Total Avg.		83.68	47.30	47.95	43.33	43.45	43.58

Examining Table 5.10, we see that TBB yields the best overall energy consumption readings. OpenMP and Threads come closely behind, with differences between them and TBB being 1.56% and 1.76%, respectively. FastFlow and SPar seem to perform more poorly, given that they are 12.06% and 12.49% inferior to TBB. The explanations for all energy consumption variations between PPIs are the same as for equivalent performance results in RO5.

Table 5.11 summarizes the average power dissipation in W for all SPBench applications across all devices. Therefore, as expected, since the sequential version uses the least amount of hardware resources, it consumes the least average power. Considering that the reduction in energy consumption from parallel and sequential versions is 54.29 to 63.53%, the increase in power dissipation between parallel and sequential versions is between 45.98 and 46.86%.

Isolating the maximum power reading during the execution of the application across all devices and calculating the average, we get that SPar peaks at 7.48 W, FastFlow at 7.01 W, TBB at 7.06 W, Threads at 7.13 W, OpenMP at 7.01 W, and sequential at 4.75 W. However, this represents at most 6.7% of the difference between them. As we discussed previously, this is only considering a single device.

Table 5.11 – Power dissipation of all SPBench applications. The maximum observed standard deviation is 0.31. For parallel versions, green and red highlight the best and worst observed results respectively.

Device	**Application**	**Average Power Dissipation (W)**					
		Seq	**SPar**	**Fastflow**	**TBB**	**Threads**	**OpenMP**
Jetson	Bzip2	3.11	5.43	5.48	5.46	5.44	5.45
	Ferret	3.09	5.62	5.63	5.66	5.75	5.70
	Lane	2.79	4.53	4.53	4.53	4.53	4.53
	Face	3.21	5.78	5.73	5.85	5.77	5.74
Raspberry	Bzip2	4.32	5.44	5.50	5.51	5.46	5.46
	Ferret	4.32	5.32	5.36	5.38	5.34	5.32
	Lane	4.55	5.55	5.60	5.57	5.56	5.49
	Face	4.55	5.48	5.48	5.46	5.46	5.45
Odroid	Bzip2	3.24	5.74	5.68	5.79	5.68	5.76
	Ferret	3.08	6.55	6.48	6.58	6.50	6.54
	Lane	3.30	6.65	6.63	6.76	6.85	6.88
	Face	3.15	6.14	6.17	6.31	6.29	6.33
Total		3.56	5.69	5.69	5.74	5.72	5.72

5.6.1 RO6.1: Analysis of memory consumption for parallel stream applications

Table 5.12 showcases the memory consumption each PPI requires to execute a parallel program with the most computational resources available in each device. Looking at the total averages, we can see the increase in percentage taking the sequential version as a base; SPar, FastFlow, TBB, Threads, and OpenMP respectively increase 90.23%, 86.88%, 72.60%, 81.10%, and 78.08%. Even the most minor memory consumption version also significantly increases the memory consumption of the sequential version. It is a factor that must be taken into account by system maintainers. The parallel versions can consume as much as 7.38 times more memory. The parallel versions in Odroid use significantly more memory because they have to sustain six parallel threads instead of 4 from Jetson and Raspberry.

Table 5.12 – Memory consumption of all SPBench applications. For parallel versions, green and red highlight the best and worst observed results respectively.

Device	Application	Memory Consumption (MB)					
		Seq	SPar	Fastflow	TBB	Threads	OpenMP
Jetson	Bzip2	11188	44984	46984	40516	43380	42860
	Ferret	99144	115548	111416	112884	116536	108660
	Lane	39156	55492	54976	47512	52320	54220
	Face	51152	90880	93288	87692	91356	89752
Raspberry	Bzip2	10732	48976	48312	45560	41184	45320
	Ferret	57524	65468	65472	64748	65184	65704
	Lane	31444	46876	46484	40160	44384	45032
	Face	43480	76368	78276	81052	82720	86892
Odroid	Bzip2	11484	84768	87288	70168	70132	65264
	Ferret	57700	129996	111520	111660	120872	115220
	Lane	37720	77292	73320	62564	62676	65216
	Face	49796	115164	118824	99548	116100	107660
Total Avg.		41710	79318	78013	72005	75570	74317

Regarding the PPIs, except TBB, all interfaces use static memory allocation. It means that they allocate memory at the start of the computation and use that until the end. When a thread requests more memory, the default language memory allocation calls are responsible for allocating it, which means they pass this responsibility to the OS. TBB has a different methodology to leverage thread-local and global pools of memory. When a TBB application requests memory, the memory allocator looks in the thread-local pool for free memory blocks. If there is no free memory in the thread-local pool, it gets a lock and a block of memory from the global pool. The global pool is shared among all threads in the application, and it is divided into fixed-size chunks that can be allocated to threads as needed. Therefore, TBB typically consumes less memory because it uses a dynamic approach.

5.7 RO7: Evaluating matrix multiplication in limited-resource hardware

Context - Matrix multiplication is a fundamental operation in many fields of study. It represents complex systems as it is a fundamental building block of many algorithms and numerical computation methods. It corresponds to the bulk of the computation of a typical machine-learning neural network. It is also essential in computer graphics and image processing.

Forces - Possibilities to increase the computational performance of matrix multiplication operations are using vectorization or SIMD instructions, multithreading, and accelerators such as GPGPUs. Furthermore, there is ongoing research with hyper-specialized accelerators for matrix multiplication that include in-memory processing or analogical computing. However, these emergent technologies face many challenges before being commercially available in most hardware devices.

The main **threat to validity** of this analysis is related to the fact that we only execute one matrix size load, which is the maximum size we could safely execute on the Odroid device. Using smaller or even greater matrix sizes broken into a series of computations to fit in the devices' GPGPU could yield different insights.

Discussion - To assess this RO, we experiment with a straightforward 2D matrix multiplication algorithm. For this algorithm, we use 1280 by 1280 matrix dimensions. Table 5.13 showcases execution time and energy consumption metrics for the Matrix Multiplication benchmark. The *Seq* version is fully optimized by the compiler. We developed a manually vectorized version of the algorithm; however, it achieves the same performance as the compiler-optimized version across all platforms. Therefore, we do not depict it in the table. The performance difference between the compiler- and manually-vectorized versions was, on average, 0.4 seconds out of 72.57 seconds. This coincides precisely with the standard deviation number.

Table 5.13 – Matrix Multiplication Metrics. The greatest standard deviations are 3.58 for the CPU and 2.21 for GPU.

Device	Version	Execution Time (s)	Total Energy Consumption (uWh)	Avg. Power Consumption (W)	Maximum Power (W)
Jetson	Seq	72.57	64.6	3.20	3.315
	CUDA	0.55	0.4	5.37	6.023
	OpenMP	14.61	18.6	4.64	4.9
Odroid	Seq	286.15	212	2.66	2.936
	OpenCL	3.98	3.4	3.31	3.468
	OpenMP	37.81	46	4.38	5.032
Raspberry	Seq	79.14	92.8	4.21	4.501
	OpenMP	36.05	52.4	5.16	5.557

Regarding Table 5.13, the most significant standard deviations are 3.58 for the CPU and 2.21 for GPU across all devices. The GPGPU versions are 132 and 72 times faster than the sequential versions in Jetson and Odroid, respectively. GPGPU versions perform much better because the Matrix Multiplication code is an ideal target for massive GPGPU parallelism. With correct implementation, it can leverage the GPGPU hardware capabilities; however, it requires a manual optimization of the kernel parameters.

Besides execution time, GPGPU parallelism yields the best total energy consumption readings. The Jetson device gets the highest average and peak power dissipation.

Compared to CPU parallelism with OpenMP, GPU parallelism gets 15.80 and 23.01% higher average and peak power dissipation. However, this differs for the Odroid device GPGPU, which has lower power than CPU parallelism in OpenMP. That is because Odroid GPU is optimized for energy efficiency, given that it is a typical smartphone GPU. On the other hand, Maxwell GPU from Jetson is optimized for general-purpose computing and can accelerate parallel processing tasks better with more available cores. The Maxwell GPU performs 621.82% better than the Mali G52 GPU.

5.8 RO8: Evaluating GPGPU parallelism for data processing applications

Context - limited-resource devices often have GPU accelerators. They are designed to handle graphics-intensive tasks much more efficiently than the device's CPU. Additionally, they are an option for general-purpose computing for neural-network inference, mathematical operations, and low-complexity robotics. Ultimately, limited-resource hardware GPUs are designed for cost-effectiveness and low energy consumption.

Forces - The GPGPU architecture favors massive parallelism, often found within data processing applications. However, these devices have more limited computing capability (simpler hardware) and available hardware, which require detailed customization. Since GPU cores are much simpler than CPU cores, developers must consider that the GPU kernel code should not be complex [7]. For instance, while high-end server GPUs have 65,536 32-bit registers per streaming multiprocessor and 32 64-bit per thread, Nvidia Maxwell (Jetson GPU) only has 192 32-bit registers and 64 64-bit registers. Additionally, the compute capabilities of Maxwell GPGPU are old (5.3), which means it does not have features and optimizations from modern GPGPUS (9.x). Furthermore, the software support for these devices is limited. Mali G52 GPU supports OpenCL, OpenGL, and Rendescript, while the Jetson Maxwell device only supports NVIDIA CUDA alternatives.

Discussion - For evaluating GPGPU parallelism, we use the CUDA implementation of all NPB applications [9]. We can successfully execute all applications of the NPB benchmark except EP, BT, and SP; they both execute, but the NPB result verification return is unsuccessful. Therefore, we omit EP, BT, and SP from our analysis. For these three applications, the output is not a number because there are not enough registers to run the code. Since the GPGPU code uses registers to store intermediate results, the code actually corrupts and returns without completing the computation. This is caused by a limitation of the GPGPU available resources, in this case, the internal registers. The solution would be to adapt the algorithm to launch smaller kernels. Similarly, we can not execute NPB applications on Odroid Mali G52 because NPB requires 64-bit precision, and Mali only supports 32-bit precision. **Limitations** are similar to the ones stated in ROs 3 and 4.

Table 5.14 depicts execution time, total energy consumption, and average power dissipation for all NPB applications on the Jetson device. In this case, we only add OpenMP as the base of comparison since it is the best-performing version as evaluated on RO3. Regarding execution time, the first factor we can observe is that, contrary to matrix multiplication, these more robust data processing applications scale less on the GPGPU. CUDA versions have an average speedup of 3.68, while OpenMP ones have 3.06. The drop in scalability is because these NPB applications perform many more operations with the memory, which hinders the application's scalability potential. Another factor that limits scalability is that NPB applications have thousands of barrier synchronization and launches up to thousands of kernels. However, GPGPU is still significantly faster; it is 11.55% up to 50% faster than OpenMP, taking the sequential as a reference.

Table 5.14 – Metrics for NPB applications with CUDA GPGPU.

Device	**Aplication**	**Execution Time (s)**			**Total Energy Consumption (uWh)**			**Avg. Power Consumption (W)**		
		Seq	CUDA	OpenMP	Seq	CUDA	OpenMP	Seq	CUDA	OpenMP
Jetson	CG	403.00	122.60	136.80	399.00	205.00	264.80	3.42	5.29	5.39
	FT	282.00	57.20	82.40	266.20	110.00	138.00	3.13	6.29	5.57
	IS	8.00	4.00	2.00	18.60	10.00	7.80	2.87	4.83	4.86
	LU	149.60	44.00	51.40	144.00	75.00	86.40	3.40	6.00	5.92
	MG	18.80	6.00	9.00	24.40	17.80	20.00	3.73	5.12	5.28
Total Average		107.68	29.23	35.20	106.53	52.23	64.63	2.07	3.44	3.38

Evaluating total energy consumption in Table 5.14, we see a similar trend to execution time. CUDA, on average, uses 68.40% less energy to perform the same set of computations than the sequential version; OpenMP uses 48.96% less. The average power dissipation is another representation of this pattern. However, examining power peaks, OpenMP gets 6.42 W while CUDA versions get up to 8.025 W. Therefore, GPGPU setups should be prepared to handle 24.98% greater power peaks.

6. SUMMARY OF FINDINGS

This chapter provides a summary of our findings and guidelines. In the remainder of this chapter, we enumerate a list of findings, briefly explain them, and relate it to the ROs that allow us to conclude them.

1 For stream processing applications, OpenMP and Threads require the developer to specify the communication queue and synchronization strategies explicitly. [RO1 and RO2]

OpenMP and Threads showcase the greatest SLOC and Halstead programmability metrics, up to 858.33% more than SPar.

2 For data processing applications, OpenMP showcases the lowest estimated complexity. [RO2]

OpenMP has 0.7 normalized Halstead, while FastFlow and TBB have 0.96. This Halstead measurement indicates that OpenMP is roughly 30% less complex than the others because its pragma directives are suitable for data processing applications. With SLOC, there are 1.61% differences.

3 Map and MapReduce data parallelism can increase performance and reduce energy consumption in limited-resource hardware. [RO3.1, RO3.2, 3.4, 3.5, and RO4]

MapReduce, with no data dependencies, can achieve 3.9 speedups using four cores, independently of PPI. With memory contention, the scalability potential is reduced, but it still achieves speedups up to 2.48. Furthermore, with data dependencies, the scalability is also limited, and the speedups are up to 2.9. It can also reduce total energy consumption by up to 31%. Ultimately, Map and MapReduce parallelism is advantageous on limited-resource hardware, even with irregular data access and direct data dependencies.

4 Devices with larger L2 cache sizes can increase parallelism scalability. [RO3.3 and RO3.5]

Irregular data access that increases L1 cache-miss rate from 3.22% up to 33.81% severely hinders parallelism efficiency. We observe that the device with double the L2 cache size achieves a speedup of 2.3 while the one with half the L2 size achieves 1.25. Therefore, L2 cache size can be the direct bottleneck factor for the scalability potential of parallelism.

5 BIG.little architecture requires special care for the thread mapping strategy. The default OS thread scheduler handles it well. [RO3.6]

Using the low-performance cores of Big.little architectures can increase performance, but efficient parallel programs must also consider this architectural feature. The OS

BIG.little aware scheduling can improve performance by up to 366% compared to a scheduling strategy that does not consider this architectural feature.

6 Parallelism can increase average hardware temperature, which impacts performance scalability. [RO3.7]

The average device temperature also increases using the maximum available hardware resources with parallelism. By keeping the average device temperature lower, we can perform from 0.9% up to 49.89% better.

7 Parallelism reduces the total execution time and energy consumption to perform a set of computations. [RO3, RO4, RO5, and RO6]

Parallelism can cause more hardware operations executed due to parallelism (i.e., synchronization, communication, and more L1 cache movements due to more cores). Conversely, lower execution time from parallelism lowers the total energy consumption by lowering the time interval that measures the idle component's energy consumption. Parallelism can reduce the total energy consumption required to perform identical computations between 54.29 and 63.53%. In contrast, the difference in throughput improvements regarding the sequential version is between 103.7 and 112.54%. Therefore, the gains in energy consumption are smaller than performance but still significant.

8 OpenMP is the best performing PPI on data processing applications. [RO3 and RO4]

`OpenMP` yields the best overall energy consumption readings. `FastFlow` and TBB perform similarly to each other (difference of 0.21%) but are, respectively, 2.09% and 1.84% inferior to `OpenMP`. Regarding execution time, `OpenMP` is best; it is 0.7% and 1.46% better than TBB and `FastFlow`, respectively.

9 Efficient parallel implementations that use more hardware resources reduce the total energy consumption but increase the power consumption and maximum power peaks. [RO4 and RO6]

Considering that the reduction in energy consumption from parallel and sequential versions is 30.36 to 31%, the increase in power consumption between parallel and sequential versions are between 65.77 and 68.05%. Additionally, the best PPIs increase the maximum power peaks by up to 4.48% between each other.

10 For stream processing applications, TBB is the best-performing PPI. [RO5 and RO6]

It achieves the lowest energy consumption (up to 10.45% better than other PPIs), latency (up to 145.08% better), and highest throughput (up to 11.75% better). The main explanations are related to TBBs dynamic approach to parallelism that handles the limited-resource domain better.

11 Parallelism increases latency in stream processing applications. [RO5]

In addition to the original computation, parallelism adds to the cost of communication, data movements, and synchronization of the computation. Comparing our experiments' average latency, the best parallel version (TBB) has 37% higher latency than the sequential version.

12 On-demand or dynamic workload scheduling can reduce latency in stream processing applications. [RO5]

Although it does not significantly degrade throughput, on-demand scheduling (queue-size 1) or dynamic scheduling can reduce latency by 132.79% up to 373.05%. It results in a more balanced computational resource distribution between stream items, resulting in shorter wait times from arrival to computation.

13 Throughput and Latency do not significantly correlate with limited-resource hardware. [RO5]

The general correlation factor latency and throughput in our experiments is 0.197. For instance, with a degree of parallelism 2, we get 659.9 milliseconds latency and 7.54 items/second throughput; with a degree of parallelism 4, we get 692.69 milliseconds latency and 13.08 items/second throughput. Even though the throughput increases by 73.54%, the latency only increases by 4.97%.

14 Blocking or busy-waiting yields lower latency and higher throughput. [RO5.1]

Blocking behavior can yield better performance because threads waiting with NOP operations can instantly retrieve and compute the stream items. Effectively, the non-blocking approach yields 68.1% less throughput and increases the end-to-end latency by 205.9%. For low-contention processing stages, meaning that it typically does not have to wait for data to be available in the queue, this increase is less relevant at 32.6%. In conclusion, blocking communication between parallel stages can significantly improve performance metrics.

15 Dynamic thread-to-computation mapping achieves lower latency and higher throughput than the static approach. [RO5.2]

For latency, the dynamic approach can get 122.95% better latency. For throughput, the dynamic approach can better handle resource contention – using the maximum number of available computational resources – improving by at least 1.72% up to 16.74%. On the other hand, the static approach achieves slightly better performance when there are exceeding available resources (ranging from 0.043287% up to 26.469%).

16 Data processing applications have a higher correlation factor between execution time and energy consumption than stream processing applications. [RO6]

The direct correlation between execution time and the energy consumption is 0.25. Data processing applications have a correlation factor of 0.51. Based on our data set, we argue that both of these correlations are significant.

17 Parallelism increases memory consumption. [RO6.1]

On average, parallelism using maximum device resources increases memory consumption by up to 90.23%. However, some parallel versions can increase memory consumption by as much as 7.38 times more memory. TBB uses the least amount of memory for parallelism because it uses an allocator logic instead of statically allocating parallel threads' memory.

18 GPGPU parallelism greatly increases performance and reduces the energy consumption of Matrix Multiplication operations. [RO7]

For limited-resource hardware, the GPGPU version can be up to 132 times faster than the sequential versions. However, it also gets 23.01% higher average and peak power consumption.

19 GPGPU parallelism is optimal for data processing applications. [RO8]

GPGPU parallelism is 11.55% up to 50% faster than the best-performing CPU version on data processing applications. It also consumes less total energy but gets 24.98% higher power peaks.

7. CONCLUSION

This Master's thesis investigated the impact and implications of parallelism on limited-resource hardware devices. This study is crucial because these devices have become increasingly more sophisticated, which imposes challenges on software developers. These challenges involve selecting the appropriate parallelism strategy, PPIs (parallel programming interfaces), and an appropriate device that meets the performance requirements of their applications. This decision process requires deep knowledge of limited-resource hardware, operating system, PPI internals, and parallelism algorithms. Therefore, this work helps to guide developers when choosing parallelism strategies or interfaces for their applications on limited-resource hardware.

To meet our goal, we proposed a set of ROs (research objectives) to guide this research in answering the leading question: *What parallelism techniques and interfaces work well for limited-resource hardware?* From that, we focused on the main part of this thesis, the experimental analysis guided by these ROs. Our experiments consider twelve applications, eight exploiting data, and four stream processing parallelism on three hardware devices with different architectural characteristics. They also evaluate programmability, performance, and consumption metrics: execution time (s), latency (s), throughput (items/s), speedup, total energy consumption (Wh), average power consumption (W), peak power consumption (W), and memory consumption (Mb). Finally, we experiment parallel versions developed using `SPar`, `FastFlow`, `TBB`, `Threads`, `OpenMP`, `OpenCL`, and `CUDA`.

We perform various experiments from our set of ROs with detailed explanations of the outcomes regarding parallelism and hardware features. We summarized these results in a series of guidelines with explanations about each conclusion pointing to the ROs deeper analysis that allowed us to conclude them. In our study, we observed that parallelism could reduce the total energy consumption required to perform identical computations between 54.29 and 63.53%. In contrast, execution time improvements are between 30.36 and 112.54%. Therefore, the gains in energy consumption are fewer than performance but still significant, a pattern that repeats in multiple ROs analysis. Additionally, we found OpenMP is the most suitable parallelism interface for data processing applications. Finally, we observed that dynamic approaches to parallelism tend to perform better, particularly TBB, in stream processing applications.

During our ROs analysis, we highlight some of the main limitations or threats to the validity of this research. Generally, they are related to the precision of our energy measurement readings and the constrained characteristics of our 12 applications. This research brought the benefits of a structured approach to evaluate many parallelism features. Additionally, its discussions bring a panorama of parallel computing that can guide parallelism developers on limited-resource hardware devices.

For future work, we can isolate and simulate characteristics with synthetic benchmarks. For instance, we can create a situation that forces an L1 cache miss and evaluate how much the L2 size affects parallelism. Similarly, we can force L2 cache miss and measure the effect of memory bandwidth on parallelism scalability. Another advancement is combining stream and data parallelism and assessing their relation on limited-resource hardware, particularly with applications that combine neural network inference and video filtering. One crucial future step is assessing distributed computing using a cluster of limited-resource devices, which is essential to the edge computing domain. Finally, another critical future research investigates programming techniques that allow robust HPC GPGPU code into limited-resource GPGPU; this includes techniques to reduce register requirements for GPGPU kernel invocations and optimization techniques.

LIST OF PAPERS

Below this paragraph, we display a list of papers published during this Master's thesis period. None of these papers relate to this Master's thesis. They were written in parallel with the Master's thesis.

- **OpenMP as runtime for providing high-level stream parallelism on multi-cores.** *The Journal of Supercomputing* [39].
- **High-level and efficient structured stream parallelism for rust on multi-cores.** *Journal of Computer Languages* [71].
- **DSParLib: A C++ Template Library for Distributed Stream Parallelism.** *International Journal Of Parallel Programming.* [52].
- **Combining stream with data parallelism abstractions for multi-cores.** *Journal of Computer Languages* [51].
- **High-Level Stream and Data Parallelism in C++ for Multi-Cores.** *Brazilian Symposium on Programming Languages.* [55]
- **Performance Data Visualization of Linux Events on Multicores.** *Simpósio em Sistemas Computacionais de Alto Desempenho.* [80]

REFERENCES

[1] Adornes, D. “A unified mapreduce programming interface for multi-core and distributed architectures”, Master’s thesis, Faculdade de Informática - PPGCC - PUCRS, 2015, 143p.

[2] Al Ghadani, A. K. A.; Mateen, W.; Ramaswamy, R. G. “Tensor-based cuda optimization for ann inferencing using parallel acceleration on embedded gpu”. In: International Conference Artificial Intelligence Applications and Innovations, 2020, pp. 291–302.

[3] Aldegheri, S.; Manzato, S.; Bombieri, N. “Enhancing performance of computer vision applications on low-power embedded systems through heterogeneous parallel programming”. In: International Conference on Very Large Scale Integration, 2018, pp. 119–124.

[4] Aldinucci, M.; Danelutto, M.; Kilpatrick, P.; Torquati, M. “Fastflow: High-level and efficient streaming on multi-core”. In: International Conference Programming Multi-core and Many-core Computing Systems, 2014, pp. 1–14.

[5] Amiri, S.; Abdi, S.; Sharifzadeh, S. “Simultaneous multiprocessing on fpga-cpu heterogeneous chips”. In: International Conference on Industrial Technology, 2021, pp. 805–809.

[6] Andrade, G.; Griebler, D.; Santos, R.; Fernandes, L. G. “A parallel programming assessment for stream processing applications on multi-core systems”, *Computer Standards & Interfaces*, vol. 84, March 2023, pp. 1–25.

[7] Andrade, G.; Griebler, D.; Santos, R.; Kessler, C.; Ernstsson, A.; Fernandes, L. G. “Analyzing programming effort model accuracy of high-level parallel programs for stream processing”. In: International Conference on Software Engineering and Advanced Applications, 2022, pp. 1–4.

[8] Andrade, H. C. M.; Gedik, B.; Turaga, D. S. “Fundamentals of Stream Processing: Application Design, Systems, and Analytics”. Cambridge University Press, 2014, 529p.

[9] Araujo, G.; Griebler, D.; Rockenbach, D. A.; Danelutto, M.; Fernandes, L. G. “Nas parallel benchmarks with cuda and beyond”, *Software: Practice and Experience*, vol. 53, Nov 2021, pp. 53–80.

[10] Bahrami, M.; Li, D.; Singhal, M.; Kundu, A. “An efficient parallel implementation of a light-weight data privacy method for mobile cloud users”. In: International Conference on Data-Intensive Computing in the Clouds, 2016, pp. 51–58.

[11] Bahri, N.; Maazouz, M.; Khemiri, R.; Masmoudi, N. “Parallel implementation of hevc encoder on multicore arm-based platform”. In: International Conference on Systems, Signals Devices, 2019, pp. 663–668.

[12] Bailey, D.; Harris, T.; Saphir, W.; Van Der Wijngaart, R.; Woo, A.; Yarrow, M. “The nas parallel benchmarks 2.0”, Technical Report, Technical Report NAS-95-020, NASA Ames Research Center, 1995, 1p.

[13] Barr, M.; Massa, A. “Programming Embedded Systems: With C and GNU Development Tools”. O’Reilly Media, Inc., 2006, 420p.

[14] Belloch, J. A.; León, G.; Badía, J. M.; Lindoso, A.; San Millan, E. “Evaluating the computational performance of the xilinx ultrascale+ eg heterogeneous mpsoc”, *Journal of Supercomputing*, vol. 77, June 2021, pp. 2124–2137.

[15] Bienia, C.; Kumar, S.; Singh, J. P.; Li, K. “The parsec benchmark suite: Characterization and architectural implications”. In: International Conference on Parallel architectures and compilation techniques, 2008, pp. 72–81.

[16] Cannizzaro, M. J.; Gretok, E. W.; George, A. D. “Risc-v benchmarking for onboard sensor processing”. In: International Conference IEEE Space Computing Conference, 2021, pp. 46–59.

[17] Chen, L.; Cui, M.; Zhang, F.; Hu, B.; Huang, K. “High-speed scene flow on embedded commercial off-the-shelf systems”, *IEEE Transactions on Industrial Informatics*, vol. 15, August 2019, pp. 1843–1852.

[18] Chronaki, K.; Moretó, M.; Casas, M.; Rico, A.; Badia, R. M.; Ayguadé, E.; Valero, M. “On the maturity of parallel applications for asymmetric multi-core processors”, *Journal of Parallel and Distributed Computing*, vol. 127, May 2019, pp. 105–115.

[19] Clemons, J.; Zhu, H.; Savarese, S.; Austin, T. “Mevbench: A mobile computer vision benchmarking suite”. In: International Conference IEEE Symposium on Workload Characterization, 2011, pp. 91–102.

[20] Coffrin, C. J. “Beyond moore’s law: Exploring the future of computation”, Technical Report, Los Alamos National Lab., Los Alamos, United States, 2019, 55p.

[21] Flink, A. “Stateful Computations over Data Streams”. Source: https://flink.apache.org/, July 2022.

[22] Garcia, A. M.; Griebler, D.; Schepke, C.; Fernandes, L. G. “Spbench: a framework for creating benchmarks of stream processing applications”, *Computing*, vol. 1, January 2022, pp. 1–23.

[23] Gilchrist, J. “Parallel compression with bzip2”. In: International Conference on Parallel and Distributed Computing and Systems, 2004, pp. 559–564.

[24] GMAP. “Parallel Application Modeling Group (Home Page)”. Source: https://gmap.pucrs.br, January 2016.

[25] Gordon, M. I.; Thies, W.; Amarasinghe, S. “Exploiting coarse-grained task, data, and pipeline parallelism in stream programs”. In: International Conference on Architectural Support for Programming Languages and Operating Systems, 2006, pp. 151–162.

[26] Gretok, E. W.; Kain, E. T.; George, A. D. “Comparative benchmarking analysis of next-generation space processors”. In: International Conference Aerospace Conference, 2019, pp. 1–16.

[27] Griebler, D. “Domain-specific language & support tool for high-level stream parallelism”, Doctoral thesis, Faculdade de Informática - PPGCC - PUCRS, 2016, 243p.

[28] Griebler, D.; Danelutto, M.; Torquati, M.; Fernandes, L. G. “Spar: A dsl for high-level and productive stream parallelism”, *Parallel Processing Letters*, vol. 27, March 2017, pp. 1–20.

[29] Griebler, D.; De Sensi, D.; Vogel, A.; Danelutto, M.; Fernandes, L. G. “Service level objectives via c++11 attributes”. In: International Conference Parallel Processing Workshops, 2018, pp. 745–756.

[30] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. “Higher-level parallelism abstractions for video applications with spar”. In: International Conference on Parallel Computing, 2017, pp. 698–707.

[31] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. “Stream parallelism with ordered data constraints on multi-core systems”, *Journal of Supercomputing*, vol. 75, July 2018, pp. 4042–4061.

[32] Griebler, D.; Hoffmann, R. B.; Loff, J.; Danelutto, M.; Fernandes, L. G. “High-level and efficient stream parallelism on multi-core systems with spar for data compression applications”. In: International Conference Brazilian Symposium on High-Performance Computing Systems, 2017, pp. 16–27.

[33] Griebler, D.; Vogel, A.; De Sensi, D.; Danelutto, M.; Fernandes, L. G. “Simplifying and implementing service level objectives for stream parallelism”, *Journal of Supercomputing*, vol. 76, June 2019, pp. 4603–4628.

[34] Guo, C.; Ci, S.; Zhou, Y.; Yang, Y. “A survey of energy consumption measurement in embedded systems”, *IEEE Access*, vol. 9, April 2021, pp. 60516–60530.

[35] Görtz, M. D.; Kühn, R.; Zietek, O.; Bernhard, R.; Bulinski, M.; Duman, D.; Freisen, B.; Jentsch, U.; Klöppner, T.; Popovic, D.; Xu, L. “Energy efficiency of a low power hardware cluster for high performance computing”. In: International Conference Informatik, 2017, pp. 2537–2548.

[36] Herlihy, M.; Shavit, N. “The Art of Multiprocessor Programming”. Morgan Kaufmann Publishers, 2008, 340p.

[37] Hoffmann, R. B. “Stream Parallelism Annotations for Autonomic OpenMP Code Generation”, Technical Report, School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil, 2020, 55p.

[38] Hoffmann, R. B.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “Stream parallelism annotations for multi-core frameworks”. In: International Conference Brazilian Symposium on Programming Languages, 2020, pp. 48–55.

[39] Hoffmann, R. B.; Löff, J.; Griebler, D.; Fernandes, L. G. “Openmp as runtime for providing high-level stream parallelism on multi-cores”, *The Journal of Supercomputing*, vol. 1, Apr 2022, pp. 1–22.

[40] Hsieh, C.; Sani, A. A.; Dutt, N. “Surf: Self-aware unified runtime framework for parallel programs on heterogeneous mobile architectures”. In: International Conference on Very Large Scale Integration, 2019, pp. 136–141.

[41] Intel. “oneapi: A new era of heterogeneous computing”. Source: https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html#gs.oik8xz, Feb 2022.

[42] ISO/IEC-14882:2011. “Information technology - programming languages - c++”, Technical Report, International Standard Organization, Geneva, Switzerland, 2011, 1338p.

[43] Jacqueline Farrell, Dick Buttlar, B. N. “PThreads Programming”. O’Reilly, 1996, 284p.

[44] Jin, H.; Frumkin, M. A.; Yan, J. C. “The openmp implementation of nas parallel benchmarks and its performance”. In: International Conference Technology Solutions, 1999, pp. 1–26.

[45] Joy-IT. “Um25c measuring instrument”. Source: https://joy-it.net/en/products/JT-UM25C, Jan 2022.

[46] Jubertie, S.; Melin, E.; Raliravaka, N.; Bodele, E.; Bocanegra, P. E. “Impact of vectorization and multithreading on performance and energy consumption on jetson boards”. In: International Conference on High Performance Computing and Simulation, 2018, pp. 276–283.

[47] Junior, J. S.; Carissimi, A. “Avaliação do consumo de energia na execução do nas parallel benchmark (npb) em processadores arm”. In: International Conference Brazilian Symposium on High Performance Computing, 2015, pp. 240–251.

[48] Khasanov, R.; Goens, A.; Castrillon, J. “Implicit data-parallelism in kahn process networks: Bridging the macqueen gap”. In: International Conference on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms, 2018, pp. 20–25.

[49] Kubiak, K.; Dec, G.; Stadnicka, D. “Possible applications of edge computing in the manufacturing industry: Systematic literature review”, *Sensors*, vol. 22, March 2022, pp. 1–7.

[50] Lee, S.-J.; Park, S.-S.; Chung, K.-S. “Efficient simd implementation for accelerating convolutional neural network”. In: International Conference on Communication and Information Processing, 2018, pp. 174–179.

[51] Löff, J.; Hoffmann, R. B.; Griebler, D.; Fernandes, L. G. “Combining stream with data parallelism abstractions for multi-cores”, *Journal of Computer Languages*, vol. 73, Dec 2022, pp. 1–21.

[52] Löff, J.; Hoffmann, R. B.; Pieper, R.; Griebler, D.; Fernandes, L. G. “Dsparlib: A C++ template library for distributed stream parallelism”, *International Journal of Parallel Programming*, vol. 50, Dec 2022, pp. 454–485.

[53] Lorenzon, A. F.; Cera, M. C.; Beck, A. C. S. “Investigating different general-purpose and embedded multicores to achieve optimal trade-offs between performance and energy”, *Journal of Parallel and Distributed Computing*, vol. 95, September 2016, pp. 107–123.

[54] Löff, J.; Griebler, D.; Mencagli, G.; Araujo, G.; Torquati, M.; Danelutto, M.; Fernandes, L. G. “The nas parallel benchmarks for evaluating c++ parallel programming frameworks on shared-memory architectures”, *Future Generation Computer Systems*, vol. 125, July 2021, pp. 743–757.

[55] Löff, J.; Hoffmann, R. B.; Griebler, D.; Fernandes, L. G. “High-level stream and data parallelism in c++ for multi-cores”. In: International Conference Brazilian Symposium on Programming Languages, 2021, pp. 41–48.

[56] Ma, T.; Bai, N.; Shi, W.; Wu, X.; Wang, L.; Wu, T.; Zhao, C. “Research on the application of visual slam in embedded gpu”, *Wireless Communications and Mobile Computing*, vol. 2021, Jun 2021, pp. 1–17.

[57] Magnussen, B. M.; Kawasumi, T.; Mikami, H.; Kimura, K.; Kasahara, H. "Performance evaluation of oscar multi-target automatic parallelizing compiler on intel, amd, arm and risc-v multicores". In: International Conference Languages and Compilers for Parallel Computing, 2022, pp. 50–64.

[58] Maheshwari, S.; Shafik, R.; Wilson, I.; Yakovlev, A.; Acharyya, A. "Repute: An opencl based read mapping tool for embedded genomics". In: International Conference Design, Automation and Test in Europe Conference and Exhibition, 2020, pp. 121–126.

[59] Marwedel, P. "Embedded System Design". Springer Cham, 2022, 433p.

[60] McCool, M.; Reinders, J.; Robison, A. "Structured Parallel Programming: Patterns for Efficient Computation". Morgan Kaufmann Publishers, 2012, 432p.

[61] Membarth, R.; Reiche, O.; Hannig, F.; Teich, J.; Korner, M.; Eckert, W. "Hipacc: A domain-specific language and compiler for image processing", *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, Jan 2016, pp. 210–224.

[62] Microsoft. "Parallel patterns library (ppl)". Source: https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl, Feb 2020.

[63] Miner, D.; Shook, A. "MapReduce Design Patterns". O'Reilly Media, 2012, 232p.

[64] Munshi, A.; Gaster, B.; Mattson, T. G.; Fung, J.; Ginsburg, D. "OpenCL Programming Guide". Addison-Wesley Professional, 2011, 648p.

[65] NVIDIA; Vingelmann, P.; Fitzek, F. H. "Cuda, release: 10.2.89". Source: https://developer.nvidia.com/cuda-toolkit, Mar 2023.

[66] OmpSs. "The ompss programming model". Source: https://pm.bsc.es/ompss, Feb 2022.

[67] OpenMP. "Open multi-processing api specification for parallel programming". Source: http://openmp.org/, Feb 2020.

[68] Ou, Z.; Pang, B.; Deng, Y.; Nurminen, J. K.; Ylä-Jääski, A.; Hui, P. "Energy- and cost-efficiency analysis of arm-based clusters". In: International Conference Symposium on Cluster, Cloud and Grid Computing, 2012, pp. 115–123.

[69] P B, H.; Anireddy, S. R.; F T, J.; R, V. "Introduction to arm processors & its types and overview to cortex m series with deep explanation of each of the processors in this family". In: International Conference on Computer Communication and Informatics, 2022, pp. 1–8.

[70] performance of intrusion detection system using OpenCL based general-purpose computing on Graphic Processing Unit (GPGPU), I. “Widianto, ahmad rinaldi and lim, charles and kho, i. eng”. In: International Conference on New Media, 2015, pp. 1–5.

[71] Pieper, R.; Löff, J.; Hoffmann, R. B.; Griebler, D.; Fernandes, L. G. “High-level and efficient structured stream parallelism for rust on multi-cores”, *Journal of Computer Languages*, vol. 65, July 2021, pp. 1–14.

[72] Pieper, R. L. “High-level programming abstractions for distributed stream processing”, Master’s thesis, School of Technology, 2020, 170p.

[73] Pop, A.; Cohen, A. “Openstream: Expressiveness and data-flow compilation of openmp streaming programs”, *ACM Transactions on Architecture and Code Optimizations*, vol. 9, January 2013, pp. 1–25.

[74] Rauber, T.; Rünger, G.; Stachowski, M. “Performance and energy metrics for multi-threaded applications on dvfs processors”, *Sustainable Computing: Informatics and Systems*, vol. 17, March 2018, pp. 55–68.

[75] Reinders, J. “Intel Threading Building Blocks”. O’Reilly, 2007, 334p.

[76] Reinders, J.; Ashbaugh, B.; Brodman, J.; Kinsner, M.; Pennycook, J.; Tian, X. “Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL”. Appress Open, 2020, 548p.

[77] Rockenbach, D. A. “High-level programming abstractions for stream parallelism on gpus”, Master’s thesis, School of Technology, 2020, 163p.

[78] Rockenbach, D. A.; Löff, J.; Araujo, G.; Griebler, D.; Fernandes, L. G. “High-level stream and data parallelism in c++ for gpus”. In: International Conference Brazilian Symposium on Programming Languages, 2022, pp. 41–49.

[79] Ruf, B.; Mohrs, J.; Weinmann, M.; Hinz, S.; Beyerer, J. “Res2tac—uav-borne real-time sgm stereo optimized for embedded arm and cuda devices”, *Sensors*, vol. 21, Jun 2021, pp. 1–37.

[80] Scheer, C.; Hoffmann, R.; Griebler, D.; Manssour, I.; Fernandes, L. “Performance data visualization of linux events on multicores”. In: International Conference Brazilian Symposium on High-Performance Computational Systems, 2021, pp. 108–119.

[81] Schmid, M.; Fritz, F.; Mottok, J. “Fine-grained parallelism framework with predictable work-stealing for real-time multiprocessor systems”, *Journal of Systems Architecture*, vol. 124, March 2022, pp. 1–10.

[82] Sensi, D. D.; Matteis, T. D.; Danelutto, M. “Simplifying self-adaptive and power-aware computing with nornir”, *Future Generation Computer Systems*, vol. 87, October 2018, pp. 136–151.

[83] Shalf, J. “The future of computing beyond moore’s law”, *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 378, January 2020, pp. 1–15.

[84] Simula, F.; Pastorelli, E.; Paolucci, P. S.; Martinelli, M.; Lonardo, A.; Biagioni, A.; Capone, C.; Capuani, F.; Cretaro, P.; De Bonis, G.; Cicero, F. L.; Pontisso, L.; Vicini, P.; Ammendola, R. “Real-time cortical simulations: Energy and interconnect scaling on distributed systems”. In: International Conference on Parallel, Distributed and Network-Based Processing, 2019, pp. 283–290.

[85] Stokke, K. R.; Stensland, H. K.; Griwodz, C.; Halvorsen, P. “Load balancing of multimedia workloads for energy efficiency on the tegra k1 multicore architecture”. In: International Conference on Multimedia Systems Conference, 2017, pp. 124–135.

[86] Storm, A. “Apache Storm”. Source: https://storm.apache.org/, July 2022.

[87] Talreja, P. V.; Durbha, S. S.; Potnis, A. V. “On-board biophysical parameters estimation using high performance computing”. In: International Conference Geoscience and Remote Sensing Symposium, 2018, pp. 5445–5448.

[88] Thies, W.; Karczmarek, M.; Amarasinghe, S. “Streamit: A language for streaming applications”. In: International Conference on Compiler Construction, 2017, pp. 179–196.

[89] Vogel, A.; Griebler, D.; Fernandes, L. G. “Providing high-level self-adaptive abstractions for stream parallelism on multicores”, *Software: Practice and Experience*, vol. 51, January 2021, pp. 1194–1217.

[90] Xie, L.; Zhang, X. “Parallel acceleration of elas on arm”. In: International Conference on Control, Automation and Robotics, 2019, pp. 235–240.

[91] Yongbon Koo, S. K. . Y.-g. H. “Opencl-darknet: implementation and optimization of opencl-based deep learning object detection framework”, *Internet and Web Information Systems*, vol. 24, Feb 2021, pp. 1299–1319.

[92] Zhou, X.; Li, R.; Zhang, P.; Liu, Y.; Dou, Y. “A pipelining strategy for accelerating convolutional networks on arm processors”. In: International Conference Communications in Computer and Information Science, 2020, pp. 519–530.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br