FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

U. PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Code Specialization for Targeting FPGAs via High-Level Synthesis Tools

Vitória Alexa Maciel Correia

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: João Cardoso

July 27, 2023

Resumo

Os "Field-programmable gate arrays" (*FPGA*) ganharam uma popularidade significativa nos últimos anos devido à sua flexibilidade, alto desempenho e eficiência energética. Isto é particularmente importante pois existe uma procura crescente de computação de alto desempenho e a proliferação de aplicações embebidas. Ferramentas de Síntese de Alto Nível, do inglês "High-Level Synthesis" (*HLS*), permitem aos programadores escrever software em linguagens de programação de alto nível, tais como *C/C++*, e gerar automaticamente circuitos de hardware que podem ser implementados em *FPGA*. Este processo oferece muitos benefícios, incluindo a capacidade de conceber circuitos de hardware de forma mais eficiente e rápida, a capacidade de aproveitar o conhecimento e as capacidades dos programadores de software, e a capacidade de alcançar um alto desempenho em *FPGA*. Embora as ferramentas *HLS* tenham melhorado nos últimos anos, numerosos estudos demonstraram que a passagem de uma linguagem de alto nível, do inglês "High-Level Language" (*HLL*), para uma linguagem de descrição de hardware, do inglês "Hardware Description Language" (*HDL*) pode muitas vezes resultar num pior desempenho em relação a implementações que são executadas num processador convencional. Como o uso de *FPGA* traz alguns benefícios fundamentais, é necessário investigar técnicas capazes de tornar este processo mais eficiente. O principal objectivo deste trabalho é expor e propor técnicas eficientes, mais especificamente, especialização de código, que é uma técnica importante para optimizar o desempenho de sistemas de hardware concebidos com *HLS* e implementados em *FPGA*. Para alcançar os nossos objectivos, foram analisadas algumas das abordagens mais recentes que incluem técnicas como a avaliação parcial, a multiversão, algoritmos de alocação de recursos, e abordagens de *co-projeto* de software e hardware. As abordagens analisadas foram capazes de demonstrar a sua eficácia quando aplicadas a um *FPGA* alvo, utilizando ferramentas de alto nível. Para facilitar a compreensão da nossa abordagem, expomos alguns dos seus principais benefícios e desafios, exemplificamos um caso de aplicação simples e real, e apresentamos uma primeira abordagem que não só permite a identificação de uma série de etapas de implementação, mas também permite a avaliação da sua eficácia.

Abstract

Field-programmable gate arrays (*FPGAs*) have gained significant popularity in recent years due to their flexibility, high performance, and energy efficiency. This is particularly important as there is an increasing demand for high performance computing and the proliferation of embedded applications. High Level Synthesis (*HLS*) tools allow programmers to write software in high-level programming languages such as *C/C++* and automatically generate hardware circuits that can be implemented in *FPGAs*. This process offers many benefits, including the ability to design hardware circuits more efficiently and quickly, the ability to leverage the knowledge and skills of software programmers, and the ability to achieve high performance on *FPGAs*. Although *HLS* tools have advanced rapidly in recent years, numerous studies have shown that moving from an High Level Languages (*HLL*) to Hardware Description Languages (*HDL*) can often result in worse performance than when implementations are run on a conventional processor. As the use of *FPGAs* brings some fundamental benefits, it is necessary to investigate techniques capable of making this process more efficient. This work's main objective is to expose and propose efficient techniques, more specifically, code specialization, which is an important technique to optimize the performance of hardware systems designed with *HLS* and implemented in *FPGAs*. To achieve our objectives, some of the most recent approaches that include techniques such as partial evaluation, multiversioning, resource allocation algorithms, and hardware software co-design approaches were analyzed. The analyzed work has shown the effectiveness of some techniques when applied to a target *FPGA* using high-level tools. To facilitate the understanding of our approach, we expose some of its main benefits and challenges, exemplify a simple, real-world application case, and present a first approach that not only allows the identification of a series of implementation steps but also allows the evaluation of its effectiveness.

Acknowledgments

The past five years of my life have been a time of growth, learning, and personal development, but they have also been defined by obstacles and failures. Knowing that this chapter of my life is drawing to a close, I cannot help but feel a feeling of accomplishment and relief as I write these acknowledgments. This project has been a significant element of this journey, and I am happy for the opportunity to work on it. As I reflect on the past, I recall the people who have always supported me. Their consistent support, encouragement, and guidance have been invaluable, and I would want to use this opportunity to acknowledge them.

I would want to acknowledge my family's unwavering support throughout my academic life. It has not been easy to be a student, especially in terms of financial and emotional support. I am so grateful for the sacrifices they have made to help me.

I am grateful to my boyfriend. During the ups and downs of this period, he has been a continuous source of support and encouragement, and I will be eternally thankful for his presence in my life.

I would want to thank my friends and colleagues, who have supported, assisted, and kept me inspired throughout the years. Their assistance and companionship were essential to my academic success.

I acknowledge my supervisor's, Professor João Cardoso, guidance, patience, and constant support over the duration of this project. His expertise and constructive feedback helped me improve my work and develop my skills.

Vitória Alexa Maciel Correia

“Start Somewhere.”

Unknown

Contents

List of Figures

List of Tables

Listings

Abbreviations

ADP	Area-Delay Product
AO	Adaptive Optimization
AP	All Programmable
BF	Belief Function
BRAM	Block Random Access Memory
CLB	Configurable Logic block
CPU	Central Processing Unit
DAE	Decoupled Access Execution
DSL	Domain-Specific Language
DSP	Digital Signal Processing
ELM	Extreme Learning Machine
FF	Flip-Flop
FPGA	Field-Programmable Gate Array
GNU	GNU’s Not Unix
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLL	High-Level Language
HLS	High-Level Synthesis
HW	Hardware
LSTM	Long-Short-Term Memory
PE	Partial Evaluation
PGO	Profile-Guided Optimization
PL	Programmable logic
PP&R	Post-place and route
QoR	Quality of Results
RTL	Register-Transfer Level
SAD	Sum of Absolute Differences
SD-VBS	San Diego Vision Benchmark Suite
SoC	System-on-Chip
SPF	Single-Precision-Float
SVM	Support Vector Machine
SW	Software
VCM	Value Counter Monitor

Chapter 1

Introduction

In this chapter, the context and motivation behind the problem addressed in this work is introduced. The main objectives of the work and the way the dissertation is organized are also described.

1.1 Context

As the complexity of digital circuits has increased over time, there is currently a tendency to focus on specific hardware applications, which allows a more efficient and faster execution of software programs on the hardware as a whole [1]. As a result, manual hardware design has become more complex, and the use of *Hardware Description Languages (HDL)*, such as *Verilog* [2] and *VHDL* [2], has become more prevalent.

Field-programmable gate arrays (FPGAs) [3] are reprogrammable hardware devices, which have come to prominence over the last few years as computational accelerators (see, e.g. [4]), and by offering a number of key benefits, such as their higher performance compared to conventional software (Central Processing Unit, *CPU*), in certain situations, at lower costs, with better energy efficiency, and by being commercially available to a wide range of users.

Thus, *FPGAs* have been growing as a new platform in different areas, which include their use in autonomous vehicle driving [5], computer vision [6], automation and safety technology at work [7], in cybernetics [8, 9], automation in vehicles [10, 11, 12], medical equipment [6], computer hardware, networking, digital communication [13, 14] and radio devices [15], bioinformatics [6], voice recognition technology [16], secure communication systems, and a variety of scientific, medical and other electronic products.

The advantages of using *FPGAs*, however, continue to be influenced by *HDLs*, used to specify their computational data path as well as their detailed programming. The use of *HDLs*, despite being able to provide efficient designs, is inefficient in terms of human efficiency, i.e., a great deal of knowledge is required by developers about hardware design, and even then, there is a high propensity for errors [4]. In this context, i.e., making the development process simpler for software programmers/developers, *High Level Synthesis (HLS)* [17, 18] tools have emerged, (see, e.g., [19, 20, 21]).

High Level Synthesis (HLS) [17, 18] provides an automated design process that, starting from an abstract behavioral specification, similar to software, generates an equivalent *register-transfer level (RTL)* structure capable of generating a circuit, which implements the input behavioral specification [22]. In other words, an *HLS* tool takes as input a function written in a *high-level language (HLL)* such as *C/C++*, and converts it to a *RTL HDL* kernel, thus being able to automatically generate an *FPGA* accelerator. *HLL* implementations can be developed more quickly and succinctly, which lowers the likelihood of programmer mistakes and increases readability and reuse of the code.

Although *HLS* tools have advanced rapidly in recent years, numerous studies have shown that switching from an *HLL* to *HDL* can often result in worse performance than when implementations are run on a conventional processor. In addition, there is input code that is, as yet, not synthesizable with current *HLS* tools, such as the use of pointers, memory management and recursion.

As an attempt to address such issues, the makers of the technology have released a collection of best practices and programming, inside this guidelines, to help programmers maximize the potential of the *HLS* tools. Researchers have shown the benefit of rewriting code, typically manually, in line with these directives [4], according to the *HLS* tool and target *FPGA*. However, there are cases where using the aforementioned guidelines has no effect or, in some cases, even worse results [4]. Furthermore, the task of manually rewriting the code is complex, time consuming, error prone, can only be performed by programmers specialized in hardware knowledge and also requires a great effort in order to optimize the synthesis, taking into account the operating frequency, parallelization and energy efficiency.

As a result, and taking into account the benefits of using *FPGA* implementations, industry and research communities are focusing on the development of tools used as supplementary steps in the design process, in particular, code optimization and code specialization techniques for *HLS* and *FPGAs*, in order to overcome these challenges.

1.2 Motivation

In order to significantly improve energy efficiency, future processors should include architectural support for customization, allowing systems to adapt to various application domains [23]. In particular, it is expected that future architectural designs will make extensive use of accelerators in hardware, such as *FPGAs*, given the opportunities they entail. However, such architectures still present numerous challenges, namely the task of refactoring, specializing and optimizing code in *HLL*, for hardware. The reality is that, despite recent advances, our understanding of how programmers perform code specialization in the real world derives mainly from research that focuses on a small number of software projects [24].

Thus, our main motivation comes from the impact of customization via code specialization in hardware and lack of information regarding the same process, taking into account the advantages introduced by *FPGA* implementations integrating specialized execution flows. It is, therefore, important to try to expose, explore and develop efficient application code specialization techniques, in a *HLL*, in particular, via *HLS* tools and targeting *FPGAs*.

1.3 Goals and Objectives

Given the aspects presented above, the main objective of this work is to propose efficient techniques for specializing the application input code, *C*, so that it becomes more suitable for the target *HLS* tool and *FPGA* and performance is improved. In some cases, multiversioning techniques are be used to guarantee the existence of very efficient accelerators in more common scenarios, but with the ability to accelerate the application in the context of other scenarios. All this depends on the identification of the most common scenarios, which can be done by profiling the code.

In summary, it is expected that this work achieves increased performance improvements by reviewing current techniques for efficient *C* code for *HLS* (as well as the possibility of discovery of new techniques), proposing efficient specialization techniques.

1.4 Dissertation's Structure

This dissertation consists of four chapters and is organized as follows:

- Chapter 2, Realated Work, presents some background regarding better-known optimization techniques, and discusses studies already carried out on several specialization techniques for code in *HLL*. In this way, based on previous studies, it is possible to obtain a perspective on existing techniques and what they involve, facilitating learning and giving an idea of which methods can be applied in our proposal, and what results to expect from them;

- In Chapter 3, Approach Description, our approach is presented, emphasising its principal stages and components. Its underlying concepts and principles are explored, along with its integration with high-level optimization and synthesis tools.

- In Chapter 4, Benchmark Analysis and Optimizations, a study of benchmarks is conducted in order to identify potential areas to apply specialization techniques. Examining the results of multiple benchmarks also allows us to identify optimization opportunities, e.g., for reducing energy consumption, increasing speed, or enhancing scalability, and establish the optimal strategy for our application.

- In Chapter 5, Experimental Results, the main goal is to describe the experimental setup and expose, analyse, and justify the results achieved by the proposed approach when applied to the benchmarks.

- Finally, in Chapter 6, Conclusions, an overview of the results obtained from the work carried out, a review of our approach, our contributions and, finally, future work are addressed.

Chapter 2

State of the Art

This chapter aims to address the topics considered relevant for understanding the problems presented in Chapter 1 and covers the research and analysis conducted on the most recent approaches and solutions related to the dissertation's main topics. It is organized into three main sections:

- Section 2.1: where a brief introduction is carried out on some important concepts for the understanding of the work to be carried out;
- Section 2.2: where work done on existing compilation techniques that specialize the *C* input application code to become more suitable for *HLS* and the target *FPGA* is discussed;
- Section 2.3: where a review on the concepts covered in the previous two points is carried out.

2.1 Background

Several key concepts that are necessary for a better understanding of the subject matter of this paper are introduced in this section.

2.1.1 Field-Programmable Gate Array

An *FPGA* [3] is a type of reprogrammable hardware device that is used in a wide range of electronic systems and devices. It consists of an array of configurable logic blocks (*CLBs*) and interconnects that can be programmed by the user to implement a wide range of digital circuits and systems. The resources of an *FPGA* are the programmable logic blocks (such as Lookup tables (*LUTs*), and *Flip-Flops*), input/output blocks, and programmable connections that make up the device. Some contemporary *FPGAs* also provide block random access memory (*BRAM*) which provides a more significant data armazenament and digital signal processing (*DSP*) components. *FPGAs* have several advantages over other types of programmable hardware devices, and in recent years, they have gained prominence as computational accelerators [4], managing to provide higher performance than conventional software (*CPU*) in several situations. The main advantages of *FPGAs*, in relation to previous ones, include: greater flexibility, higher speed, determinism, energy efficiency, re-programmability, parallelism, which in turn allows to improve performance, and customizability, considering that *FPGAs* can be customized for specific applications, allowing users to optimize their designs for their specific needs. For these reasons, they are commonly used in a wide range of applications, including high-performance computing, communications [13, 14], networking [8, 9], aerospace, military [12], medical [6], automotive [11], and industrial systems [7].

FPGAs are, however, still limited by the complexity and detail of the *HDLs*, used to specify their computational data path.

2.1.2 High Level Synthesis

HLS tools [17, 18] arise in the context of making the use of *HDLs* a simpler development process for software programmers/developers. *HLS* [17, 18] is a design methodology for automatically generating hardware implementations of programs written in high-level languages like *C* or *C++*. *HLS* tools analyze the program and automatically generate specialized hardware implementations of the program's functions and algorithms, which can be synthesized into hardware circuits using *HDLs*, such as *VHDL* or *Verilog*.

In recent years, *HLS* has been increasingly studied by industry because it offers several advantages over traditional design methodologies, such as *RTL* design, higher productivity, quality, portability, and design exploration. Even though *HLS* tools have come a long way in recent years, more research and development is still needed in this area so that the change from a high-level language to an *HDL* is more efficient and leads to better performance than when implementations run on a regular processor.

2.1.3 Code optimizations

Code optimization is the process of improving the performance, efficiency, and reliability of a program by modifying its code without changing its functionality.

In the context of *HLS*, code optimization involves using *HLS* tools to automatically optimize the hardware implementations of a program that are generated by the *HLS* tool. This can involve a wide range of techniques, including improving the algorithms used in the program, reducing the amount of memory and other resources used by the hardware implementation, and improving the utilization of the *FPGA* and other hardware resources. The specific optimizations that are used depend on the target *FPGA* and the goals of the optimization.

There are several types of code optimizations that can be performed in the context of *HLS*. Some of the main types of code optimizations in *HLS* include:

- Loop pipelining [25]: This technique involves breaking a loop into smaller parts and running each part in parallel, which increases the amount of parallelism;

- Loop unrolling [26]: This technique involves duplicating the body of a loop and then unrolling it. This reduces the overhead of the loop control logic;
- Data flow optimization [27]: This technique involves identifying data dependencies in the hardware implementation and then rearranging the hardware loops to improve data flow. It reduces the amount of data that needs to be stored in memory;
- Function inlining [28]: This optimization technique involves replacing a function call with the body of the function itself, thereby reducing function call and return overheads and allowing the compiler to perform further optimization of the code within the function;
- Data Reuse [29]: Data reuse is a code optimization technique that involves reusing the same data in multiple computations rather than storing and accessing it multiple times, reducing the amount of memory and other resources that are used by the program, and allowing hardware circuits to operate more efficiently.

2.1.4 Code Specialization

When developing a function, programmers may sometimes declare a variable that turns out to be "quasi-invariant" [30] or even constant. In such cases, the variable can be assigned to a literal value as soon as it is appropriate. This may eliminate the overhead added by computation and helps the compiler detect the variable value at compile time. This last advantage, can facilitate the implementation of other optimizations such as constant propagation, constant folding, dead code elimination, loop unrolling, and pipelining [31, 28].

There are two types of specialization: dynamic and static specialization. Static specialization makes use of data that is expected to be used frequently, while dynamic specialization uses the actual values at runtime [31]. Code specialization is performed more effectively at runtime due to the unavailability of input values. The main difficulties related to specialization are: 1) detecting which variables are interesting to specialize and their values, and 2) minimizing the number of variables to specialize while maintaining a generic code and having good performance results [32].

Code specialization in the context of *HLS* is a technique that allows designers to write more general, high-level code, such as *C* or *C++*, for their digital circuits and then use *HLS* tools to automatically generate lower-level, more efficient *RTL* code that can be used to program target hardware, such as *FPGAs*. This approach can help speed up the design process and make it more flexible, as it allows designers to focus on the high-level functionality of their circuits without worrying so much about the low-level details. In addition, code specialization can help improve the performance and energy efficiency of the resulting hardware by allowing *HLS* tools to explore a larger space of possible *RTL* implementations and choose the one that is best suited for the target hardware platform.

There are several types of code specialization techniques that can be used in the context of *HLS*. Some examples of code specialization techniques in *HLS* include: parameter specialization, loop specialization, path specialization, architecture specialization, mode specialization, data type specialization, memory specialization, and task specialization. The specific techniques that are used depend on the target *FPGA* and the optimization objectives.

2.1.5 Value Profiling

Value profiling [33] is a technique for analyzing and collecting data regarding the distribution of a program's inputs or values during its execution (during runtime) in order to improve its performance.

This is achieved by repeatedly executing the program and collecting data on the inputs, variables, and expressions whose values are calculated by the program. This data can then be used to direct the optimization process by identifying numbers and areas of the code that are often used and run, responsible for the majority of the computation, and where performance improvements are anticipated to have the most impact. With this data, it is possible to develop code variations that are optimized for the most frequent inputs or values.

This technique could have a substantial impact on the context of code specialization in *HLS* tools for *FPGAs*, since the collected information can be used to drive the code specialization process by optimizing the implementation for the most frequent values. By focusing the implementation on the most frequent values, the design can be optimized for both performance and area, resulting in a more effective *FPGA* implementation. Moreover, value profiling can help uncover performance bottlenecks in the code, which can then be solved via code specialization.

2.1.6 Partial Evaluation

According to [34], partial evaluation of a computer program is, by definition, the "specialization of a general program based on its operating environment into a more efficient program".

Partial evaluation is a technique in which the computation is evaluated with respect to the value of a particular operand. That is, instead of treating an operand as a variable, its value becomes "fixed" [35, 28]. This can be useful in situations where a function or program is called multiple times with the same arguments, as it allows partial evaluation to be performed once the resulting simplified function or program is used in subsequent calls, saving computational time and resources.

Based on the example in [34] and [28], let us imagine a function, f, that takes two arguments, a and b, i.e., $f(a,b)$, as shown in listing 2.1.

```
int f(int a, int b) { return b * (a - b) + a; }
```

Listing 2.1: Partial Evaluation Example: original function

Now suppose we want to use this function several times in our program, but always with the same value for b, for example, $b = 2$. In this case, we can do a partial evaluation of the function f to generate a new specialized function g (see listing 2.2):

```
int g(int a) { return 3 * a - 4 ;}
```

Listing 2.2: Partial Evaluation Example: specialized function

This transformation can save computational time and resources since function g becomes simpler and more efficient than function f for this specific set of input arguments.

In the context of *HLS*, partial evaluation is a technique that can be used to optimize the generated *RTL* code by simplifying it for a particular set of input arguments. This can be useful because the high-level code that is input to an *HLS* tool is often more general and abstract, and may not make use of all the input arguments available in a particular application.

2.1.7 Multiversioning

Multiversion is a well-known technique in code optimization for generating code that can adapt to a changing execution context: at compile-time, numerous versions of the original code are generated as the consequence of distinct optimization modifications. When the resultant code is executed, decisions are taken at runtime to select the appropriate version based on particular criteria, such as input data or runtime behavior. There are numerous approaches to implement multiversion, depending on the system's unique requirements and limits.

When all future execution contexts are known, static multiversion, i.e. several versions generated during compilation, can be efficient. These execution contexts may be associated with the hardware execution platform on which the code is actually executed or with target attributes of the code that influence the validity or performance of the generated versions. This method becomes unworkable when execution contexts are largely unknown at compile time and cannot be predicted. The same holds true for specialization and unpredictable values. The generation of runtime versions of code where execution contexts are evidently understood might be one solution to these limits. Nonetheless, this method requires multiple time-consuming steps: profiling, analysis, transformation, and just-in-time compilation. Runtime multiversioning can be especially advantageous due to the fact that parameter values are frequently unknown at compile time and only discovered during execution.

2.2 Related Work

Code specialization, in the context of *HLS*, is a technique used to optimize the hardware implementation generated for a specific target device. It involves selecting and customizing hardware components and optimizing their interconnections based on the requirements and constraints of the target device. In recent years, there has been an increase in the use of code specialization techniques in *HLS* for *FPGA* targets. These techniques aim to improve the performance and resource utilization of the generated hardware implementation and can be applied at various stages of the *HLS* flow, including instruction selection and other optimizations.

This section is mainly aimed at discussing the work presented on some of the most recent approaches in compilation techniques for code specialization using *HLS* and a target *FPGA*, which include the use of machine learning-based optimization methods and the incorporation of domain-specific knowledge in the optimization process. Additionally, techniques will be addressed, which, despite presenting a view directed towards software performance improvement and not particularly on *HLS* tools or the use of hardware devices, may have utility for the development of new ideas for solution formulation and technique development.

2.2.1 Related approaches using HLS with a target FPGA

HeteroRefactor

Lau et al. [36] make an innovative end-to-end proposal, known as *HeteroRefactor*, which combines dynamic invariant analysis, automated refactoring, and selective offload for *FPGAs*.

HeteroRefactor focuses mainly on the following *HLS* refactoring techniques: rewriting a recursive data structure to a finite-sized array, reducing the bit-width of integers, and tuning variable-width floating-point operations. These techniques are based on the expectation that dynamic *a-priori* analysis improves *FPGA* synthesizability, resource efficiency, and input-dependent offloading can guarantee correctness. For rewriting recursive data structures, the *HeteroRefactor* implementation is based on the ROSE [37] compiler framework. Whereas to reduce the bit-width of integers, it uses an approach based on the dynamic invariant detection tool, *Daikon* [38] using *Kvasir* [38] as a *C/C++* front-end. The implementation of these first two types of refactorings is performed in a similar way for selective offloading, using a guard condition check. For floating-point operations, on the other hand, a dynamic analysis was performed that provides a probabilistic guarantee that the loss of precision is within a certain limit, this is because unlike the bit-width reduction of integers, when the bit-width of floating point variables is reduced, it can lead to a loss of precision.

HeteroRefactor consists of three main steps, which occur in the following order:

1) After a programmer implements his kernel code, code is then executed on existing tests where it is verified the existence of *FPGA*-specific dynamic invariants, i.e., bit-width needed for floating-point and integer variables, stack size and recursive data structures;

2) From the information obtained in 1), it automatically refactors the kernel in order to make programs *HLS* compatible, and at the same time, optimize the clock frequency and resource usage of the accelerator.

3) If the input fits the dynamic invariant, selectively offloads computation from the *CPU* to the *FPGA*, ensuring the preservation of correctness behavior. Otherwise, it keeps the computation on the *CPU*.

To evaluate this approach, ten diverse and suitable for different situations programs were used as benchmarks. The kernels were generated from *C/C++* and targeted a *Xilinx Virtex UltraScale+XCVU9p FPGA* [39] on a *VCU1525* reconfigurable acceleration platform [40]. Using *Vivado Design Suite 2018.03* [41] the refactored programs are synthesized for *RTL*.

Based on experimental results, it was concluded that *HeteroRefactor* is able to automate the transformation (no need to change the code) of a recursive program into an *HLS* compatible version, identifying the empirical limit for the size of the recursive data structure, avoiding the effort of an experienced *FPGA* programmer. Furthermore, for recursive programs, the results showed that this approach had a great impact in reducing the resource usage of the target *FPGA*.

AnyHLS

It is common for programmers to write pragma-annotated *C/C++* programs to define a hardware architecture of an application. Given that, each hardware vendor uses its own set of specific pragmas and the challenge of portability between different vendors arises. Additionally, the difficulty of using pragmas in a modular way (as they are resolved by the preprocessor) or performing proper abstractions, contributes to the difficulty of using existing *HLS* languages.

Thus, in [42] is presented *AnyHLS*, which unlike existing *HLS* approaches, focuses mainly on the challenges mentioned above by synthesizing *FPGA* designs in a modular and abstract way. To do so, it resorts to exploiting concepts of partial evaluation and programming language features (higher-order types and functions). *AnyHLS* is built on top of *AnyDSL* [43]. AnyDSL provides partial evaluation, which means there is no need to modify the compiler when adding support for a new application domain as developers can design custom control structures. *AnyDSL* is a compilation framework able to create high-performance, domain-specific libraries (*DSLs*) [44], and provides the *Impala* [34] language. *Impala* allows programmers to evaluate their programs partially at compile time by controlling the partial evaluator through filters. In addition, it provides syntactic sugar for invoking certain higher-order functions/generators , which are very powerful when combined with partial evaluation. In this paper, using the *Impala* functional language and its partial evaluation it has been possible to realize the abstractions required for *FPGA* synthesis in the form of a library. The partial evaluation is necessary to combine the abstractions and remove the respective overheads. After optimization of the user application based on a library of abstractions and partial evaluation, the *AnyDSL* compiler [43] synthesizes the optimized *HLS* code (*C++* or *OpenCL*) from a given functional description of an algorithm. Subsequently, the generated code is input to the selected *HLS* tool.

To evaluate this approach, seven programs were used as benchmarks, diversified and according to the image processing application domain. The generated *HLS* codes were compiled using *Intel FPGA SDK* [45] for *OpenCL 18.1* [45] and *Xilinx Vivado HLS 2017.2* [41] targeting *Cyclone V GT 5CGTD9 FPGA* [46] and *Zynq XC7Z020 FPGA* [47], respectively. The results of *AnyHLS*, after *PP&R* (post-place and route), were compared with the results of other domain-specific approaches, *Halide-HLS* [48] and *Hipacc* [49].

This approach, besides solving the challenge of portability (avoiding, in the source code, vendor-specific pragmas) and modularity, shows that the combination of high-order abstractions and partial evaluation is powerful enough to make the design of a library with different *HLS* compilers as targets. Furthermore, when compared to other *DSLs* (*Hipacc* and *Halide-HLS*), *AnyHLS* was able to generate equally efficient designs without creating a backend compiler entirely, outperforming them in terms of productivity.

FPGA-based SVM acceleration

Tsoutsouras et al. [6] present a systematic, two-level methodology and prototype structure for developing high-performance *FPGA*-targeted hardware Support Vector Machine *SVM* accelerator projects. The initial step of optimization in the presented methodology entails restructuring the *SVM* source code to expose greater levels of data- and instruction-level parallelism. According to the authors of the article, this level of optimization involves "reorganizing the data and instruction relationships in the source code to enable a greater degree of parallel execution." To expose more parallelism in the code, this may require techniques such as loop pipelining, loop unrolling, array split, and array reshaping, among others. This level of optimization is crucial because it permits the *SVM* code to be tailored to the exact hardware platform on which it will be deployed. By exposing additional parallelism in the code, the hardware accelerator can use the parallelism existing in the *SVM* algorithm and potentially achieve greater performance than with a non-specialized version of the code.

At level one of optimization, they also concentrate on minimizing hardware resource use. In order to investigate the design space of multiple hardware architectures and configurations that can implement *SVM* algorithms, they use an *HLS* tool. They employ a set of parameters, including area, power, and performance, to assess the efficiency of the developed hardware projects.

At level two of optimization, they use the results from level one to select the most efficient hardware design in terms of execution time for the *SVM* algorithms. In addition, they explain strategies for fine-tuning the hardware's design to increase its efficiency.

This methodology targets *SVM*-based system-on-chip systems, notably *Zynq* [47], which offers an *ARM Cortex-A9* and *SVM* fabric. All layers of accelerator optimization are performed using the *Vivado-HLS* tool [41], which gives the user directive-based control over the synthesis process. A case study of an ECG-based arrhythmia detection flow proved the efficiency of this methodology for producing efficient accelerator designs.

In general, this methodology can result in very efficient *SVM* accelerator implementations that deliver great performance with less area than manually optimized designs. Therefore, the application of this methodology can be advantageous for code specialization in the context of hardware acceleration.

2.2.2 Partial Evaluation

As described in Section 2.1.6, partial evaluation is a technique that optimizes the performance of a program by evaluating it with some of its inputs predetermined.

Dong et al. [50] suggested a new hierarchical activity recognition method based on the theory of belief functions (BFs) [51, 52]. The theory of belief functions provides a mathematical framework for the representation and manipulation of uncertain data. It is a non-probabilistic approach to uncertainty based on the concept of a mass of beliefs, which expresses the degree of confidence that an event will occur. The theory of belief functions is especially relevant for dealing with complex systems and circumstances when knowledge is incomplete or ambiguous. Using approximations or heuristics, partial evaluation approaches can be utilized to optimize system performance by decreasing the computational burden of belief function calculations. This can be accomplished by precalculating and storing certain features of the belief function computations in a table or other data structure for reuse at runtime. The method provided in [50] employs a long-short-term memory (*LSTM*) model [53] to classify specified activities and a confusion matrix to determine similarities between each pair of activities. On the basis of hierarchical clustering [54], the activities are then structured into a hierarchical structure, and an Extreme Learning Machine (*ELM*) model [55] is trained for each non-leaf node. During the testing phase, a novel, efficient, tree-based matching rule is proposed to combine the findings of all ELMs that have been trained. *UCI Smartphone* [56] and *mHealth* [57] datasets are used to evaluate the hierarchical activity recognition algorithm presented. The results demonstrated this method surpassed all advanced algorithms.

The authors in [44] offer a technique for shallow embedding domain-specific languages (*DSLs*) into a host language via online partial evaluation [58]. The partial evaluation technique is intended to enhance the efficiency of *DSLs* implemented with shallow embedding by providing optimized host language code that can be performed directly, as opposed to interpreting or compiling the *DSL* code each time it is executed. The authors implement this technique by combining static analysis with runtime analysis. They begin by conducting a static analysis of the *DSL* code to determine which areas can be partially assessed and which cannot. This is accomplished by examining the *DSL* code's types and dependencies at compile-time. This partial evaluation technique analyzes the *DSL* code at runtime in order to generate the optimal host language code. The generated code for the host language is then compiled and executed, avoiding the need to interpret or compile the *DSL* code each time it is executed. Using several case studies, the authors demonstrate the efficiency of this technique and discuss its challenges and limitations, such as the difficulty of statically analyzing the *DSL* code to determine which portions can be partially evaluated and the overhead of generating and compiling the optimized host language code.

2.2.3 Multiversioning

Lazcano et al. [59] designed a memoized speculative loop optimizer, a technique that aims to improve code by detecting computationally costly loops and executing numerous versions of these loops in parallel. The objective is to discover the quickest version of the loop for the current hardware setup and record the result for future reference. A multiversion runtime technique was created as an extension of the Speculative Polyhedral Loop Optimizer (Apollo) framework [60, 61] to enable specialization and execution of many versions. The proposed multiversion runtime system is divided into two distinct phases: training and operation. During the training phase, modifications are made to the loop core, and the optimal version is chosen depending on runtime. Based on the kernel settings, this version is kept and indexed (memoized). In the operational phase, the saved version is re-released anytime the identical parameters are detected, resulting in optimal performance without the need for additional analysis or transformation. If a kernel with changed parameters is released, a new training phase is initiated.

Using heuristic methods, the method presented in [62] selects a minimal collection of representative optimization alternatives (function versions) for multiversioning structures while avoiding performance loss across accessible datasets and code size explosion. The method employs a novel mapping mechanism that utilizes decision trees or machine learning-based rule induction techniques to efficiently select the optimal code version at runtime depending on dataset features and to minimize selection overhead. This allows the production of static self-adjusting binaries or adaptive libraries that may modify their behavior and adapt to changing runtime settings without the need for complex recompilation structures. Thus, this methodology might efficiently pick the best versions of code at runtime while minimizing selection overheads, resulting in increased performance and code size efficiency, as well as higher adaptability and flexibility in comparison to standard static compilation techniques. The authors evaluated their strategies using the Open64 compiler [63].

"Multiversioning with Dynamic Access Edition" (*SMVDAE*) is the term given by the authors to a novel strategy proposed in [64]. This innovative approach aims to convert sequential code automatically at compile time in order to enable energy-efficient execution via decoupled access execution (*DAE*) [65]. This multiversion technique generates many access versions of a software application, ranging from lightweight versions that may be less efficient at preloading cached data to more complicated versions that may be more efficient but incur greater overhead. The optimal access version is chosen dynamically at runtime, thus bypassing the restrictions of static analysis. The main advantages of *SMVDAE* are the fact that it can effectively explore the space of optimized access versions at runtime, since multiversion is performed statically, and the fact that dynamic selection of the optimized version incurs minimum overhead. The method also contains a mechanism for identifying memory access patterns to determine which access phases are most effective for each task, which is useful for optimizing code in general.

"Cooperative Profile-Guided Optimization" [66] is a new technique that includes a new multiversion approach and code specialization method that can improve the performance of *GPUs* in interactive applications. This technique combines Profile-Guided Optimization (*PGO*) and Adaptive Optimization (*AO*). Cooperative *PGO*, like *AO*, functions most effectively in the context of a just-in-time (*JIT*) compilation virtual machine, as it is based on the dynamic construction of instrumented program variations for the collecting of profile data. *Cooperative PGO*, like *PGO*, employs previously gathered runtime profiles to guide future compilations. With *Cooperative PGO*, programmers do not need to guess which inputs are representative for their applications. *Cooperative PGO* considerably increases the number of versions that may be compared, which is the primary distinction between this method and conventional multi-versioning methods.

A methodology for implementing a *RegionSeeker* framework together with its extension Multiversioning has been developed in [67], which adds to the advancement of the *HW/SW* co-design field. The authors present effective solutions to the problem of choosing automatically which portions of an application should be synthesized to *HW*, given a particular area budget. The RegionSeeker-identified accelerators regularly beat those obtained by data stream-level methods and solely evaluating function-level possibilities, regardless of application size or area limits. The multiversioning strategy expands the initial set of possibilities by offering numerous optimized variants. Given an initial set of *HW* accelerators, i.e. a set of regions determined using the RegionSeeker framework, several variants of each area with the same functionality can be generated. Each version may utilize one or a mix of the optimizations determined by the writers.

Cardoso et al. [28] discuss the use of autotuning techniques to maximize the performance of a program automatically by modifying parameters or factors that affect program performance. This can be accomplished offline (i.e. statically at compile-time) utilizing heuristics and profiling to guide parameter settings for compile-time optimizations and code transformations, or online (i.e. dynamically at runtime) utilizing a combination of static analysis and profiling to create models and make code execution decisions. In the context of multiversion approaches, autotuning can be used to generate and tune several versions of a program for specific hardware and software platforms, thereby enhancing the performance of the program on those systems. Autotuning can be used within the framework of code specialization to generate and optimize automatically specialized versions of a program for particular input data or use cases. It is crucial to note that the efficacy of autotuning might vary depending on the application and platform being used, and that autotuning can be time-consuming and demand large computational resources.

2.3 Summary

In recent years, research into code optimization techniques for *HLS* and *FPGA* targets has increased. One of the primary advantages of code specialization for *HLS* and *FPGA* targets is that it permits designers to tailor the performance of their hardware systems for certain applications and/or workloads. This can be particularly beneficial for applications with demanding performance requirements, such as those found in multimedia, communications, and high-performance computing. Code specialization can also be used to reduce the size and complexity of hardware systems, which is useful for applications with resource limitations such as power, area, or cost.

In Section 2.2, we discussed the work presented on some of the most recent compilation techniques for code specialization using *HLS* and targeting *FPGAs*, as well as techniques that, despite presenting a vision aimed at enhancing software performance, can be useful for the development of new ideas for formulating solutions and developing other approaches. After analyzing the

Table 2.1: Related approaches main characteristics

Approach	Specialization techniques	Included features	Main results
[36]	HeteroRefactor	Dynamic invariants, automated refactoring, selective offloading for FPGA	Recursive Programs: reduction of programmer effort, and BRAM by 83%, increase of frequency by 42%; Integers: reduction of FF by 25%, look-up-tables by 21%, BRAM by 41% and DSP by 52%; Floating Point: With an acceptable loss of precision of 10^-4 and 95% confidence, reduction of FF by 61%, LUT by 39% and DSP by 50%.
[42]	AnyHLS	Functional abstractions with Partial evaluation	With Xilinx Vivado: less resource usage, latency and better performance than other DSL approaches; With Intel FPGA SDK for OpenCL: AnyHLS achieves similar performance to Hipacc, but outperforms it in multikernel applications, which means AnyHLS optimizes interkernel dependencies better.
[6]	FPGA-based SVM acceleration	Systematic two-level methodology: Code Restructuring for HLS (Loop, Memory Partitioning and ILP) and Design Space Exploration of HLS Directives	SVM accelerator designs achieving latency gains of up to 98.78 % in respect to Vivado-HLS default optimized solution; Using ECG analysis and Arrhythmia detection, target Zynq programmable SoC utilizing the optimized SVM accelerator design outperforms pure software implementations in numerous single or dual core target platforms, achieving speedups, which range from 10× up to 78×.
[59]	Runtime Multi-versioning and Specialization	Optimizing transformations are applied on-the-fly (polyhedral optimizing and parallelizing loop transformations); Multi-versioning runtime; Memoization optimizer;	On both systems, significant speedups are obtained for the majority of loop kernels and issue sizes. A minor slowdowns can be seen for a few kernels, though.
[62]	static multiversioning approach with dynamic version	Traditional iterative compilation techniques (using collective optimizations); heuristic; machine learning techniques	Was possible to effectively prune a large number of versions optimized for different datasets. This is achieved without considerable performance loss nor code size explosion. This techniques can improve the overall performance of static programs or libraries with low run-time overhead.
[64]	Multiversioned Decoupled Access-Execute	SMVDAE	Significantly reduces the energy expenditure of irregular or memory-bound applications and even yields slight performance boosts. Overall, achieves over 20% on average energy-delay-product (EDP) improvements (energy over 15% and performance over 5%) across 14 benchmarks from SPEC CPU 2006 and Parboil benchmark suites, with peak EDP improvements surpassing 70%.
[66]	Cooperative PGO	Mix between Profile-guided, adaptative optimizations for code specialization and Multi-versioning dynamically	A PGO that exploits likely zeros is particularly effective, achieving an average speedup of 5%, with a maximum speedup of 15%, over a highly-tuned baseline.
[67]	RegionSeeker MuLTiVersioning	Compiler-driven methodology	RegionSeeker MuLTiVersioning offers 1.8x speedup over the total run-time of the jpeg application compared to SW execution and up to 65x speedup on the parts of the computation that are synthesized into HW. Moreover, compared to single version approaches it achieves 2-6x speedup over the run time of the selected regions.
[50]	Hierarchical Activity Recognition Based on Belief Functions Theory (Partial Evaluation)	Hierarchical activity recognition method based on Belief Functions (BFs) theory: first use Long-Short Term Model (LSTM) and then determine the similarities between each pair of activities through a confusion matrix	Compared with several algorithms, the results prove that the approach has a superior performance over all the advanced algorithms.
[44]	Online partial evaluator for continuation-passing style languages	Continuation-passing style (CPS)-based language Impala together with a novel online partial evaluator	Produces highly specialized and efficient code for CPUs as well as GPUs that matches the performance of hand-tuned expert code.

approaches in Section 2.2, it was possible to summarize their most significant characteristics in twelve aspects. Table 2.1 illustrates the three characteristics considered most essential for this initial phase of the project. This strategy permits the comparison of the aforementioned methods and, as a result, the research of our solution.

Through the examination of these methods, it was feasible to determine some of their limitations and underutilized capabilities. Using high-level tools, [42, 36, 6] were able to demonstrate their efficiency when applied to a target *FPGA*. However, despite the increase in research in this field, there is still a lack of information addressing the application of specialization targeting hardware. The majority of available information is derived mostly from studies focusing on a small number of software projects [59, 62, 64, 66, 67, 28, 50, 44]. Both partial evaluation and multiversioning techniques [59, 62, 64, 66, 67, 28, 50, 44], employed in software development projects, could be utilized to enhance the performance of hardware circuits developed using *HLS* tools. However, implementing these techniques can be challenging and may necessitate a substantial amount of design effort in order to obtain desirable outcomes. The requirement to strike a balance between performance and resource use is another important obstacle. *FPGAs* have limited resources that must be managed carefully to achieve good performance, and *HLS* tools must balance the use of these resources with the need to achieve performance improvements. Additionally, it is essential to support a wide range of apps and platforms, which can be difficult due to the varying performance needs and resource limits of each application.

In conclusion, the current state-of-the-art indicates that there is still more work to be done in this field; consequently, one of our aims will be to expose, study, and develop effective code specialization approaches using *HLS* tools and targeting *FPGAs*.

Chapter 3

Approach Description

As described in Chapter 1, the primary objective of this study is to propose an approach for specializing the input application code so that it is better suited for generating more efficient hardware accelerators via *HLS* and targeting *FPGAs*.

The purpose of this Chapter is to describe the proposed strategy, emphasising its principal stages and components. The approach's underlying concepts and principles are explored, along with its integration with high-level optimization and synthesis tools such as *Xilinx's Vitis HLS* [68]. The steps involved in the approach's workflow are presented, from benchmark selection and analysis to the development and evaluation of optimized versions.

Section 3.1 describes the adopted workflow, as illustrated in Figure 3.1, as well as the categories of tools that may be beneficial in each process phase. In Sections 3.2 through 5.5, the proposed tools for each phase are described in terms of their functionalities and specific contributions. Section 3.8 concludes with a very simple practical example that serves as motivation and demonstrates just a part of the potential offered by the application of this approach.

With this information, it is hoped to provide a comprehensive review of the proposed approach, emphasising its significance and benefits in code optimization and the generation of efficient hardware accelerators.

3.1 Workflow

In order to achieve the primary objectives of this project, it is essential to propose a strategy that allows not only the identification of a series of steps for implementing our approach but also the evaluation of its efficiency. The diagram representing the workflow is presented in Figure 3.1. The workflow is divided into three main phases: collection of the necessary information to select the required techniques for specialization and multiversion; application and analysis of the techniques in an appropriate environment; and, finally, evaluation of the techniques on a *FPGA*.

The first part is aimed at gathering all the necessary information to be able to carry out the work and to be able to choose appropriate benchmarks. A good selection of benchmarks can help to understand the trade-offs between various optimization techniques and resource usage

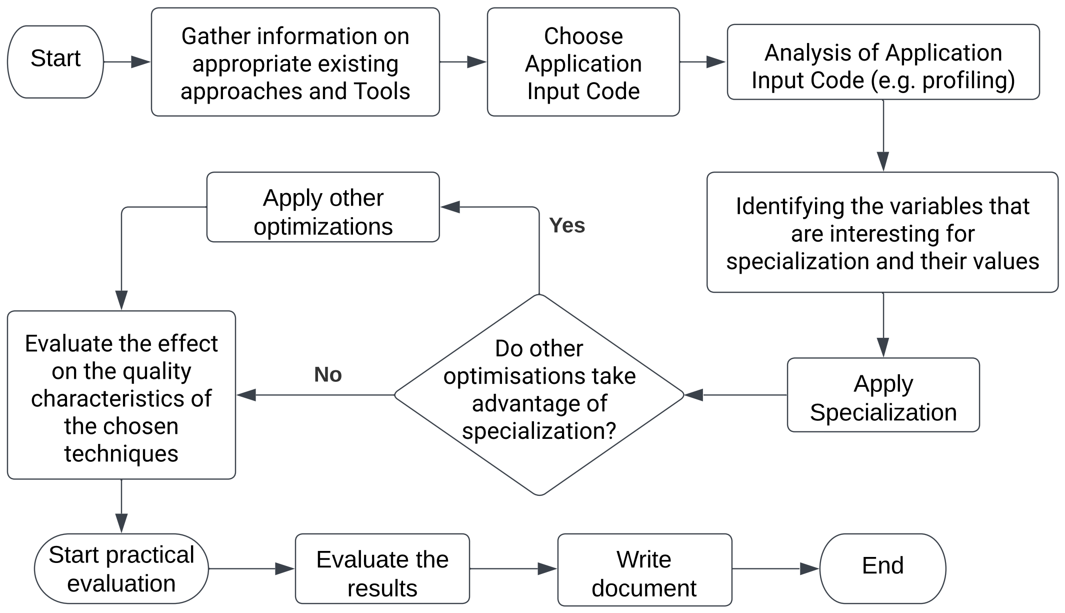

Start
Gather information on appropriate existing approaches and Tools
Choose Application Input Code
Analysis of Application Input Code (e.g. profiling)
Identifying the variables that are interesting for specialization and their values
Apply Specialization
Do other optimisations take advantage of specialization?
Yes
Apply other optimizations
No
Evaluate the effect on the quality characteristics of the chosen techniques
Start practical evaluation
Evaluate the results
Write document
End

Figure 3.1: Approach workflow Diagram

strategies, as well as identify the most efficient procedures for certain applications and platforms. The second part starts by analysing the input application code, each benchmark, and identifying the key performance indicators of the code to help determine whether specialization techniques can be used. Each change made should ensure that the behaviour of the program is maintained. To carry out the task of analysis, profiling tools and libraries capable of code variables monitoring were used. Considering the chosen specialization technique, other optimizations and other techniques like multiversion can be applied. Tools capable of detecting similarities between different versions of benchmarks could be useful to help in the choice of which versions should be grouped to create a multiversion version. The chosen *HLS* tool, *Vitis HLS 2022.2* [68], is then used to assess the impact of the modifications made. This process is systematically replicated for different solutions. It is also essential to select a suitable reference to properly compare and evaluate the performance of each version of the algorithms. In the next phase, the same tests are implemented in practice on the *PYNQ*™*-Z2* [69, 70] board, and the results are reported and analysed.

In general, the optimization of certain functions and variables should be based on a thorough examination of the target system's characteristics and the algorithm's requirements. This strategy gives a starting point for potential performance improvements, but it is essential to adapt it to the specific circumstances and do performance tests to evaluate the effects of optimizations.

3.2 The San Diego Vision Benchmark Suite: Map Disparity

It is essential to select an appropriate benchmark for properly compare and evaluate the performance of systems and algorithms. A good selection of benchmarks can aid in comprehending the trade-offs between various optimization techniques and resource usage methods, as well as identifying the most efficient procedures for certain applications and platforms. We use benchmarks from *The San Diego Vision Benchmark Suite (SD-VBS)* [71] for our study, which is a collection of computer vision applications extracted from the computer vision domain.

SD-VBS consists of nine benchmarks, however we have chosen only one, *Map Disparity* [71], to act as a support for this project's initial phase. The *Map Disparity* technique computes depth information for scene objects based on a pair of stereo pictures captured from slightly different places. It is commonly employed in robotic vision systems for applications such as cruise control, pedestrian identification, and collision avoidance. The implementation of *Map Disparity* is based on Depth Perception, and computes dense disparity by operating on each pixel of the image. From the perspective of a programmer, disparity possesses program properties such as regular memory accesses, and predictable workload, making it a programmer-friendly parallelization method whose performance is limited only by the ability to bring data to the processor.

3.3 Taskmark (working title)

Taskmark (working title) [72] is an ongoing collection of task-intensive applications optimal for heterogeneous *CPU* and *FPGA* systems, maintained by Tiago Santos and coworkers and accessible at [72]. The primary objective of the authors is to provide pure source code and compilation instructions, as well as embedded input data (i.e., header files), so that the code can be analysed and synthesised with *HLS* tools with minimal effort. Taskmark provides the K-Nearest Neighbours benchmark (*kNN*) for this activity. *kNN* is a well-known algorithm for classification and regression tasks in supervised machine learning. It classifies new data based on how closely it resembles previously classified training data. *kNN* is extensively used in numerous disciplines, including medicine, security, and speech recognition. Due to its computational complexity in tasks such as pattern recognition and image classification, it is frequently employed for benchmark analysis and performance optimization with large datasets. Its simplicity facilitates code optimization strategy comparisons.

3.4 Gprof and Gprof2dot

In this work, the GNU tool *Gprof* [73] and the *Gprof2dot* script [74] are used to analyse and profile the benchmarks' behaviour. These tool is well-known and is used to evaluate the performance of *C/C++* programs.

Gprof, a component of the *GNU* toolkit (*GNU* Profiler), plays a crucial role in the collection of program runtime data. It generates a report that displays the amount of time spent on each function of the code. Using metrics such as cumulative time, number of function calls, and other pertinent data, it is possible to determine which sections of the code occupy the most runtime and which critical areas are most likely to benefit from optimization.

Gprof2dot can be used to make performance analysis more visual and intuitive, making it simpler to comprehend the profiling data and identify potential areas for improvement using the profiling data generated by *Gprof*. It converts the *Gprof* data into *DOT* graph format (callgraph). This graphical representation facilitates comprehension of the program's function hierarchy and main call paths.

In conclusion, *Gprof* and *Gprof2dot* provide a clear view of the performance of the code, making it simpler to comprehend the relationships between functions and more effectively target improvements.

3.5 Value Counter Monitor

The ability to monitor variables and intermediate values in calculations is a very useful technique during the analysis phase of the program, particularly when there are a large number of variables and complex calculation operations involved.

Monitoring variables and intermediate values is applicable in a variety of contexts and can facilitate the identification of code performance challenges. In machine learning, computer vision, and image processing, for instance, complex computations are frequently comprised of intermediate variables whose values are typically constants and zero. By monitoring and identifying these variables, we can simplify the operations involving these zero values, accelerate the code, and save a significant amount of computational resources by avoiding superfluous operations in these regions.

The article "*PGZ: Automatic Zero-Value Code specialization*" [75] introduces a technique, which is based on the concept of monitoring intermediate values and variables in calculations and identifying the ones that are zero. In this technique, a program is executed multiple times with various inputs, and its variables' values are monitored and recorded during each execution. Using such information and zero value specialization, it is possible to identify patterns and determine which intermediate values are zero. This implementation has demonstrated significant performance enhancements across a variety of program types, demonstrating that this technique is an useful instrument for code optimization.

In this project, we utilise the "monitors" library previous developed in the *SPECS* group by our colleague Pedro Pinto. This library includes a "Value Counter Monitor" (*VCM*) that counts the number of occurrences of one or more values assigned to a specific variable by using a hash table. Figure 3.3 illustrates how the library works. When the *vcm_init* function is called, a new "*vcm*" (value counter monitor) and hashtable are constructed for the user-specified variable "a" during program execution. Then, anytime the *vcm_inc* function is called for the new "*vcm*," it means that the "a" variable gets a value. This value could or could not be contained in the hashtable. If the value is already present, it is used as a key in the table to access the number of its occurrences

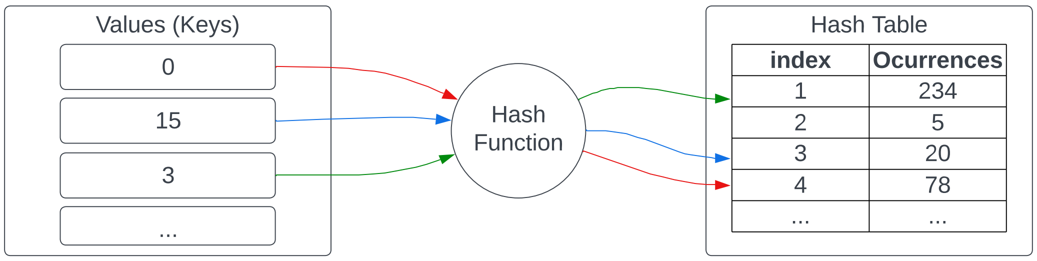


Figure 3.2: "monitor" library hashtable

and increase it by one. If the value does not yet exist in the table, it is either ignored if the table is fully populated or a new entry is created for that value. In the end of the program, a histogram showing the occurrence of each value assigned to that variable is generated. Figure 3.2 represents the diagram that exemplifies how the library hashtable works.

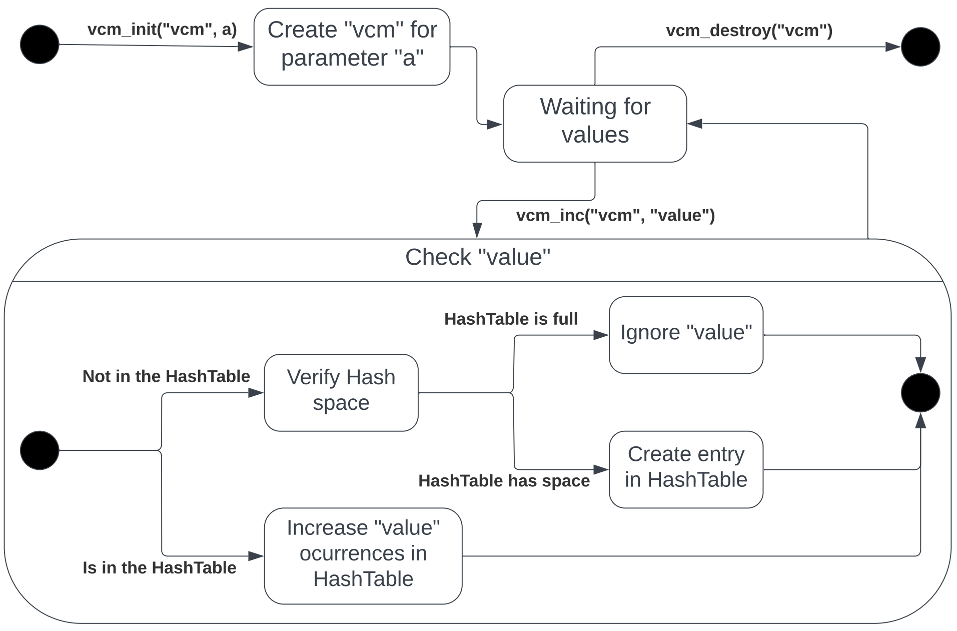


Figure 3.3: "monitor" library state diagram

In this project, we extended the "monitors" library. In the initial version of the "monitors" library, the user could specify the maximum number of distinct values that could be stored to a variable in the hashtable. However, when the variable under evaluation has a number of occurrences of different values higher than the maximum limit of occurrences that could be stored and counted in the table, only the first occurrences, corresponding to the maximum number defined, are counted. The remaining occurrences are discarded.

To improve this functionality, it was decided to incorporate the substitution policies *FIFO* (First-In, First-Out), *LRU* (Least Recently Used), *LFU* (Least Frequently Used), *MRU* (Most Recently Used), and *RANDOM* in the library. The user may choose which one to employ for each monitored variable or parameter. These substitution policies enable the "monitors" library to manage the table's limited space in an efficient manner, ensuring that occurrences of more relevant values are retained while less relevant values are substituted. When the maximum limit is reached, the *FIFO* policy replaces the oldest value inserted into the table with the occurrence of a new value. With the *LRU* policy, it is possible to replace the value that has been accessed the longest, i.e., the value that has been accessed the least recently, when the maximum limit is reached. The *LFU* policy prioritises the removal of values with fewer occurrences by replacing the least frequently used value when the maximum limit is reached. In contrast to the *LRU* policy, the *MRU* policy replaces the most recently accessed value when the utmost limit is reached. Using the *rand()* function, the *RANDOM* policy selects a random value between 0 and the utmost number of elements that can be present in the table, and then substitutes the corresponding value in the table.

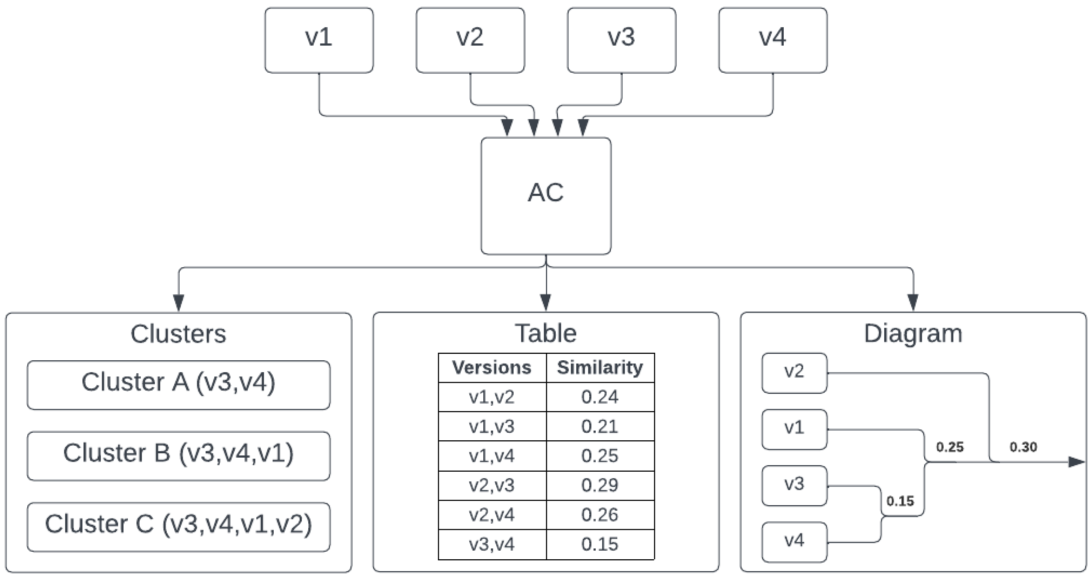
In addition to the substitution policies, two parameters have been added that can determine the total number of values assigned to the monitored variable, and the total number of values substituted in the table. These parameters enable a more accurate comparison of the various substitution policies and their efficiency in table management.

It was also made possible for the user to select a range of values from which to determine the possible values associated with the monitored variable. This feature enables the user to establish more precise and specific data analysis criteria. As seen in Section 5.4 of Chapter 5, this is especially useful when working with large data sets in which the variable being monitored can take on a broad range of values. By limiting the scope to the relevant values, the user saves time and effort interpreting the results, resulting in a more efficient analysis and a more thorough comprehension of the data under consideration.

3.6 Multiversion: Relationship with the degree of similarity

During the development of this study, we explored if there is a correlation between the degree of similarity between different individual versions and the results obtained when applying a version that employs the multiversion technique of these versions. By investigating this relationship, we hope to determine whether the similarity between versions of an algorithm can serve as a reliable indicator of expected performance when using the multiversion technique, as well as how to efficiently select and combine versions. If we observe that similarity between versions consistently yields similar results when employing the multiversion technique, this will strengthen the dependability and utility of this strategy as an algorithm optimization technique.

This idea was inspired by the research conducted in [76]. The research employs data mining in software code to identify patterns applicable to the design of hardware cores. However, because of the large number of samples and the imprecise definition of code features for mining, the researchers propose the use of three techniques: Normalised Compression Distance, Neighbour



Versions	Similarity
v1,v2	0.24
v1,v3	0.21
v1,v4	0.25
v2,v3	0.29
v2,v4	0.26
v3,v4	0.15

Figure 3.4: AC results example

Joining, and the Fast Newman algorithm. These techniques are combined to produce *DAMICORE*, a novel approach for data mining in code repositories. *DAMICORE* works with various forms of code representation and can identify significant similarities at the level of source code. Experiments demonstrated that *DAMICORE* can indicate the fusion of software kernels, resulting in smaller *FPGA* cores than when each kernel is implemented separately. This research contributes to the advancement of knowledge in the field of software implementation in hardware by emphasising the significance of version similarity analysis and its connection to efficient *FPGA* resource utilisation. It is important to note, however, that although the results obtained are encouraging, additional research is required to deepen the understanding of this relationship and investigate other variables that may affect *FPGA* resource utilisation in various software versions.

We use the *AC* [77] tool to conduct this research. *AC* is a tool for detecting instances of plagiarism in source code written in languages such as *C*, *C++*, *Java*, *Python*, and *VHDL*. AC uses the Normalised Compression Distance as the primary similarity measure. This tool, which can detect similarities between different versions of source code, can be used to compare the specialized versions generated by the multiversion technique with the original versions and determine the degree to which they are similar. In addition, it provides visualisation capabilities that enable a more in-depth analysis of similarities discovered. The results can be presented in tables, colour tables, and graphs, including clusters and trees, as illustrated in Figure 3.4. This facilitates the identification of patters and relationships between code versions, aids in validating the results and comprehending the effects of the performed optimizations.

3.7 Design Flow

Several design stages are required to convert *C* language input code into *FPGA* designs. The main stages are represented in Figure 3.5.

Following the necessary guidelines and constraints for *HLS*, the first stage is to write the *C* language code that implements the desired functionality. The *C* code is then compiled using an *HLS* tool. *HLS* converts *C* code into an equivalent hardware description, mapping *C* constructs into logical elements and hardware structures suitable for *FPGAs*. *HLS* provides a number of optimization options that can be used to enhance performance and resource utilisation on the *FPGA*. It is essential to perform functional verification on the transformed code after *HLS* to ensure that it retains the same functionality as the original *C* code.

The following stage involves generating the *IP* using the *HLS* tool. The *IP* is a description of the hardware block produced by *HLS*. It encapsulates the implemented functionality and is reusable for future *FPGA* applications. Using the generated *IP*, the *Xilinx* design tool *Vivado* is used to implement the FPGA design. *Vivado* enables the configuration and integration of *IP*, the definition of timing and routing constraints, and the compilation of the *FPGA*-programmable bitstream file. After implementation and routing are completed successfully, *Vivado* generates the bitstream file. The bitstream contains all of the information required to configure the *FPGA* for the final design, including the logic element parameters and interconnections. The final step is to program the generated bitstream into the target *FPGA*, *PYNQ*™*-Z2* board. After programming the target *FPGA*, it is necessary to conduct tests to ensure that the program functions as anticipated.

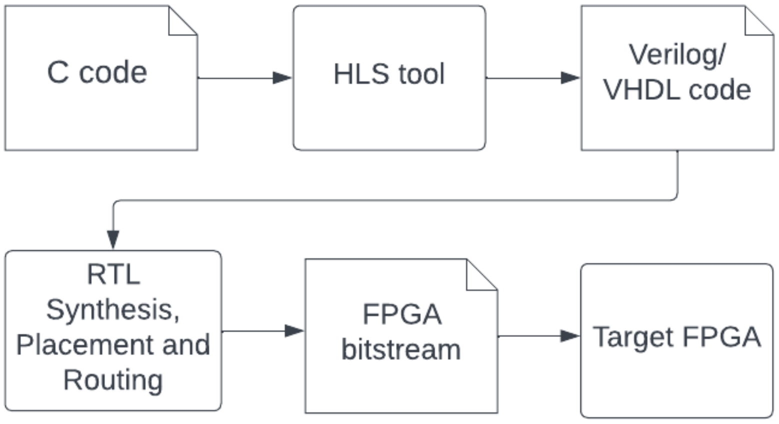


Figure 3.5: Design Flow Main Stages

3.7.1 Vitis HLS 2022.2

Xilinx's Vitis HLS [68] is a *HLS* framework that automatically generates hardware implementations of code written in high-level languages, like *C* or *C++*. We use version *2022.2* of *Vitis HLS*, which brings a number of enhancements and additional functionality over earlier iterations.

Overall, *Vitis HLS 2022.2* is an effective tool for creating hardware accelerators and other specialized hardware circuits from high-level applications. The performance, effectiveness, and dependability of hardware designs may be enhanced by a variety of its features and capabilities.

3.7.2 Vivado 2022.2

Vivado [78] is a suite of tools developed by *Xilinx* for the design and development of *FPGA*-based digital systems. It offers a variety of features and functionality to assist designers in the implementation of complex hardware solutions.

The *Vivado* suite consists of a number of tools that span various stages of the design flow, ranging from *HDL* code synthesis to physical implementation and *FPGA* device programming. *Vivado* is compatible with a variety of *Xilinx FPGA* devices, including the *Virtex* family, *Artix*, *Kintex*, and *Zynq* series, which combine programmable logic with processors. In addition, *Vivado* offers a variety of analysis, debugging, and optimization tools, such as the *Vivado* Power Analyzer for power consumption analysis, the *Vivado* Logic Analyzer for real-time debugging, and the *Vivado IP* Catalogue, which provides a vast selection of predefined IP blocks.

This project uses the *Vivado 2022.2* release.

3.7.3 PYNQ™-Z2 board

In this work, we use *PYNQ-Z2* [69, 70], a development board based on the *Xilinx Zynq-7000 All Programmable System-on-Chip (AP SoC)*. It is meant to be used with the *Python* programming language to construct high-level applications that take advantage of the hardware acceleration capabilities of the *Zynq AP SoC* [79]. The *Xilinx Zynq-7000 All Programmable System-on-Chip (AP SoC)* utilized in the *PYNQ-Z2* development board comprises a dual-core *ARM Cortex-A9* processor. The *ARM Cortex-A9* is a high-performance and low-power processor capable of regulating the operation of the board, communicating with external devices, and processing data. The *PYNQ-Z2* board is frequently used for a variety of applications such as machine learning, image and video processing, and embedded system development. The *Zynq AP SoC* provides a versatile architecture for installing custom hardware accelerators, making it particularly well suited for applications needing high-performance computing or hardware acceleration.

3.8 Motivation Example: Pow Function

Consider the math.h library's *pow()* function. Listing 3.1 shows a function using *pow()* to calculate the value of a number *a* raise to a number *b* and return its result to a variable *c*.

The first step in specializing this function is to identify the dynamic variables as well as the static variables that might be used for the specialization. Following the identification of the variables, where the specialization may be applied, three types of potential situations emerge:

1. Situations where there is only one specialization since the *a* and/or *b* take only one value in the execution of the program;

```c
#include <math.h>
double calculate_pow(double a, double b)
{
    c = pow(a,b);
    return c;
}
```

Listing 3.1: Example of using the pow() function in the C programming language

2. Situations where several specializations may exist given that *a* and/or *b* take several values in the execution of the program;

3. Situations where several specializations may exist since *a* and/or *b* take more often some values in the program execution and there is the need to ensure that they can take any value.

In Table 3.1 several scenarios are presented, which most falls under situation 1), and their specializations resulting expressions. It is important to be aware of the typical tradeoff between size and speed. In this case, the size of the residual program grows along with the value of the input *b*, and for a large *b*, specialization may be undesirable.

In relation to situations *2)* and *3)*, scenarios may arise where a variable has multiple values, for instance, $b = 1$ sometimes and $b = 3$ other times. In such cases, a multiple versioning strategy, known as *Multiversioning*, can be used to generate several versions of a function or algorithm automatically, each one optimized for a different set of circumstances or operational requirements. Therefore, based on the precise entries and the program's operational requirements, the appropriate version of the function to use is chosen automatically at runtime. In [28], a possible scenario within this situation is exemplified, in which a conditional test, *if()*, is used to check the program state and call the appropriate filter, taking into account the existing version.

Several versions of the *pow* function, each with a distinct specialization, can be derived from the scenarios presented in Table 3.1. The functionalities of a few of the conceivable versions are displayed in Table 3.2. These versions include specialized versions (*SPEC*), versions with *multiversion* (*MV*), and versions where we know the function's parameters are single-precision float (*SPF*). In Section 5.2.1 of Chapter 5, the results analysis after *HLS* of each version's latency and speedup are presented in detail in Tables 5.2 and 5.3; resources used and percentage comparison with the generic version are presented in Tables 5.4 and 5.5; and values corresponding to the

Table 3.1: Examples of scenarios that may occur in pow() function specialization

Scenario	**Input Value**	**Specialized Code**
1	*a = 1*	*c = 1*
2	*b = 1*	*c = a*
3	*b = 3*	*c = a * a * a*
4	*b = 0.5*	*c = sqrt(a)*
5	*b = -0.5*	*c = inv(sqrt(a))*
6	*a,b* in single-precision-float	*c = powf(a,b)*
7	*a,c* in single-precision-float, *b = 0.5*	*c = sqrtf(a)*

```c
#include <math.h>
double calculate_pow(double a, double b)
{
    if (b == 1) c = a;
    else if (b == 3) c = a*a*a;
    else c = pow(a, b);

    return c;
}
```

Listing 3.2: Example of the application of specialization with multiversioning with respect to pow() function

Area-Delay product and respective percentage comparison with the generic version are presented in Table 5.6.

In short, when comparing the specialized versions to the generic version, a reduction or maintenance of latency cycles are observed. Versions *2* and *5* do not necessitate additional latency cycles, resulting in substantial accelerations. The versions *v3_SPEC_b3*, *v6_SPEC_ab_spf*, and *v7_SPEC_ac_spf_b05* outperform the generic version by a factor of *6.62x*, *2.26x*, and *4.53x*, respectively. In the majority of specialized versions, resource consumption is reduced by more than *60%*. This suggests that the program could profit from specialization and justifies the use of the *multiversion* optimization strategy. Additionally, the *multiversion* versions demonstrated faster or comparable performance to the generic version. The *multiversion* strategy incorporates the advantages of generic and specialized versions, enabling the selection of the version that is most effective for each set of conditions.

Although, for instance, the expression $c = pow(a,b)$ makes the code more understandable for a programmer, it might be detrimental to performance due to the additional function call overhead. Additionally, executing the operation inline rather than using the function enables the compiler to perform additional optimizations, including loop unrolling. It is very simple to identify the $pow()$ specializations, but not all of them are valid for this transformation, and if the algorithm has a large number of them, invalid filtering events may become problematic.

This example, despite being very simple and generic, provides a number of specializations features that can occur in a variety of applications.

3.9 Summary

This chapter explored the proposal for an approach to specialize application input code in order to generate more efficient hardware accelerators. The developed approach is described, emphasising its major phases and components and exploring into the concepts and principles underlying them.

The adopted workflow is described in Section 3.1. It is divided into three major phases: information acquisition, application and analysis of techniques in an appropriate environment, and evaluation of techniques on an *FPGA*. The appropriate selection of benchmarks is emphasised as a crucial aspect of comprehending trade-offs and identifying the most effective procedures.

Table 3.2: Versions of the pow function and their transformations.

Version	Specialization	Functionality
v1_SPEC_gen	Generic	c = pow(a,b)
v2_SPEC_b1	SPEC: b = 1	c = a
v3_SPEC_b3	SPEC: b = 3	c = a * a * a
v4_SPEC_b05	SPEC: b = 0.5	c = sqrt(a)
v5_SPEC_a1	SPEC: a = 1	c = 1
v6_SPEC_ab_spf	SPEC: a, b single-precision-float	c = powf(a,b)
v7_SPEC_ac_spf_b05	SPEC: a, c single-precision-float, b = 0.5	c = sqrtf(a)
v8_MV_b1	MV: b = 1 || Generic	if(opt == 1) c = a; else c = pow(a,b);
v9_MV_b3	MV: b = 3 || Generic	if(opt == 1) c = a * a * a; else c = pow(a,b);
v10_MV_b05	MV: b = 0.5 || Generic	if(opt == 1) c = sqrt(a); else c = pow(a,b);
v11_MV_b1_b3	MV: b = 1 || b = 3 || Generic	if(opt == 1) c = a; else if(opt == 2) c = a * a * a; else c = pow(a,b);
v12_MV_a1	MV: a = 1 || Generic	if(opt == 1) c = 1; else c = pow(a,b);
v13_MV_spf_b05	MV spf: b = 0.5 || spf Generic	if(opt == 1) c = 1; else c = pow(a,b);

Taskmark's *kNN* benchmark (working title) and *SD-VBS*'s Disparity benchmark were chosen for this endeavour. The chapter also discusses the use of tools such as *Gprof* for code analysis and profiling, as well as the "monitors" library for monitoring variables and intermediate values during program analysis. The technique used in [75] is mentioned as a strategy for identifying null values and optimizing code based on these patterns. A modification was also made to the "monitors" library in order to save time and effort when interpreting the results, make the analysis more effective, and provide a more thorough comprehension of the data under consideration.

A correlation between the degree of similarity between various individual versions and the results obtained when a version employing the multiversion technique of these versions is applied is also evaluated. This study is conducted with the assistance of *AC*, a tool for detecting instances of plagiarism in source code. This research was motivated by [76], in which data extraction from software code was used to identify patterns pertinent to the design of hardware cores.

The design flow is presented in Section 3.7, which is an essential stage in the process of optimizing code for *FPGAs*. It is comprised of a series of sequential stages that transform the source code into an *FPGA*-implementable hardware design. After creating specialized versions for each benchmark, the next stage is to test them using the *Vitis HLS* and obtain the results. This procedure is repeated systematically for various solutions. *Vitis HLS* enables the exploration of various architectures and configurations in order to acquire an optimized design, as well as the specification of performance and area constraints to direct the hardware synthesis.

The next stage, following the synthesis, is the *Vivado*-based physical implementation of the design. When executing the design flow, it is essential to take into account the characteristics and limitations of the *PYNQ™-Z2* board, which was selected for use. The chapter concludes with a straightforward practical example that demonstrates some of the potential offered by the approach's application. In Chapter 5, the results are presented and analysed.

Chapter 4

Benchmark Analysis and Optimizations

Benchmarking is one of the most important way for engineers to figure out how well hardware and software systems work. Through benchmark analysis, it is possible to measure how well an algorithm works in different situations, such as with different inputs and different sizes of data. This enables developers to discover and optimize important code regions, resulting in more efficient and scalable solutions.

Benchmark analysis is particularly significant in the context of code specialization for *FPGAs* utilizing *HLS* tools, since it enables the identification of critical code sections that can then be targeted for specialization. It is possible to determine memory access patterns, data sizes, and other code properties that directly impact performance. This information is vital for selecting the best specialization approaches and adjusting the synthesis settings of the tool. Consequently, benchmark analysis is a vital step in ensuring that the specialized code is both resource and performance efficient.

In this chapter, a study of benchmarks is conducted in order to identify potential areas to apply specialization techniques. Examining the results of multiple benchmarks also allows us to identify optimization opportunities, for e.g. for reducing energy consumption, increasing speed, or enhancing scalability, and establish the optimal strategy for our application.

Section 4.1 analyses the *kNN* benchmark, which is mentioned in Section 3.3 of Chapter 3. The transformations that resulted in the various versions of the algorithm are presented, as well as the *Vitis HLS* tool directives that were applied to each of them. The same approach takes place in Section 4.2.1 for the *SD-VBS* disparity algorithm, which was previously introduced in Section 3.2 of Chapter 3.

4.1 k-Nearest Neighbors

K-Nearest Neighbors (*kNN*) [80] is a supervised machine learning algorithm primarily used for classification and regression problems that can, e.g., classify a new instance of data based on its proximity to previously classified training data. Due to its effectiveness, ease of comprehension, and implementation, this algorithm is used in many fields, including medicine, finance, marketing, security, and speech recognition, among others. This algorithm's use requires distance calculations and result ordering, which can be computationally intensive. As a result, this algorithm is a good candidate for benchmark analysis, as optimizing its performance is applicable to a variety of applications due to its frequent use with large datasets. Its simplicity facilitates benchmark analysis and comparisons between various code optimization strategies.

4.1.1 kNN Analysis

In this study, we use the *kNN* algorithm from *Taskmark* (working title) [72], which is a collection of heavy-duty applications optimal for *CPU* and *FPGA* systems with heterogeneous processing architectures.

The *kNN* utilized incorporates the *kNN_Predict* function. This function was regarded as the top function in Vitis *HLS* due to its computationally intensive characteristics. The *kNN_Predict* function contains the training data *training_X* and training labels *training_Y*, as well as a query datapoint *queryDatapoint* and the minimum and maximum values for each feature in the training data. It also contains the *kNN* implementation parameters *n_features*, *n_testing*, *n_training*, *k*, *bestDistances*, *bestPointsIdx*, and *n_classes*. Invoking *kNN_MinMaxNormalize*, the function normalizes the query reference point using *min-max* normalization (scaling the feature values to the range [0, 1]). *kNN_InitBest* is then used to initialize *bestDistances* and *bestPointsIdx*, setting all *bestDistances* values to *DBL_MAX* and all *bestPointsIdx* values to -1. Next, for each training datapoint, the function computes the Euclidean distance between the query datapoint and the training datapoint, and uses *kNN_UpdateBestCaching* to update the *bestDistances* and *bestPointsIdx* sets if the distance is smaller than any of the current *bestDistances* values. The function then uses *kNN_VoteBetweenBest* to total the occurrences of each class among the k-nearest neighbors and returns the class that appears most frequently. The *kNN_UpdateBestCaching* implementation reduces the number of iterations required to discover the two largest values in *bestDistances* by utilizing caching. This is accomplished by keeping note of the indices of the worst and second worst distances observed thus far and updating these values as necessary while iterating the array.

4.1.2 kNN Optimizations

In this section, a number of versions of the *kNN* algorithm resulting from transformations aimed at code specialization and optimization for execution on *FPGAs* using *HLS* tools are analysed. The key distinctions between each version in terms of optimization strategies are highlighted. It is discussed how each version attempts to enhance performance by reducing the number of latency cycles, decreasing resource consumption, and enhancing the use of specific functions.

All the different simulation scenarios utilised by the *kNN* algorithm are listed in Table 4.1. This study focuses on the scenarios *WI_K3_F* and *WI_K20_F*. Consequently, the transformations conducted are tailored to the specific characteristics of these scenarios.

Table 4.1: Possible scenarios for kNN algorithm.

Scenario ID	Dataset	Data type	#Features	#Samples Training	#Samples Testing	K	Accuracy
WI_K3_F	wisdm	float	43	4336	1082	3	68.02%
WI_K3_D	wisdm	double	43	4336	1082	3	68.02%
WI_K20_F	wisdm	float	43	4336	1082	20	68.76%
WI_K20_D	wisdm	double	43	4336	1082	20	68.76%
GA_K20_F	GA	float	100	8004	1996	20	50.50%
GA_K20_D	GA	double	100	8004	1996	20	50.50%
GB_K20_F	GB	float	100	40002	9998	20	51.17%
GB_K20_D	GB	double	100	40002	9998	20	51.17%

Original version (v1)

The original version of *kNN* utilised includes a high degree of specialization in terms of the parameters used, such as *N_TRAINING*, *N_TESTING*, *N_FEATURES*, *N_CLASSES*, and *K*, which were previously declared as constants. These constants differ solely based on the scenario specified before to program execution, guaranteeing that the algorithm works in general, regardless of scenario.

Generic version (v2)

As previously stated, the first version of the *kNN* algorithm already had a high level of specialization when compared to other *kNN* algorithms. As a result, it would not be an appropriate basis for comparison with other *kNN* versions. Based on this thought, an alternative, more generic version was created in which the parameters were not previously established as constants. Listings 4.1 and 4.2 depicts an example of a change made from the original form to the generic version, respectively.

All subsequent versions of the *kNN* algorithm were derived from the generalised version. This strategy allows for the exploration of various configurations and the adjustment of parameters for each individual version. Because the base version was adaptable, it was feasible to implement transformations and optimizations with greater freedom, adjusting the algorithm to diverse contexts and individual requirements. This concept of producing versions derived from the generalised version provides a systematic and comparative approach to evaluating the performance and efficiency of each *kNN* algorithm modification.

Listing 4.1: Original code (v1)

```
void kNN_PredictAll(DATA_TYPE training_X[N_TRAINING][N_FEATURES],
    CLASS_TYPE training_Y[N_TRAINING],
    DATA_TYPE testing_X[N_TESTING][N_FEATURES], CLASS_TYPE testing_Y[N_TESTING],
    DATA_TYPE min[N_FEATURES], DATA_TYPE max[N_FEATURES])
{
    int i;
    for ( i = 0; i < N_TESTING; i++)
    {
        testing_Y[i] = kNN_Predict(training_X, training_Y, testing_X[i],
        min, max);
    }
}
```

Listing 4.2: Generic code (v2)

```
void kNN_PredictAll(DATA_TYPE *training_X, CLASS_TYPE *training_Y,
    DATA_TYPE *testing_X, CLASS_TYPE *testing_Y, int k, DATA_TYPE *min,
    DATA_TYPE *max, int n_features, int n_testing, int n_training,
    double *bestDistances, int *bestPointsIdx, int n_classes)
{
    int i;
    for ( i = 0; i < n_testing; i++)
    {
        testing_Y[i] = kNN_Predict(training_X, training_Y,
        &testing_X[i*n_features], min, max, n_features, n_testing, n_training,
        k, bestDistances, bestPointsIdx, n_classes);
    }
}
```

Version with specialization $k = 3$ (v3)

The parameter K determines how many near-training examples should be considered when classifying a new data point based on the *kNN* algorithm. By employing a specialization of K, it is possible to optimize the resulting circuit, thereby decreasing the number of resources required to implement the circuit, which leads to a decrease in circuit size, latency, and power consumption. This can be especially important in embedded systems with limited resources, where resource efficiency is a significant concern. Nevertheless, specialization to a particular K value may reduce circuit flexibility (e.g., adapting to different inputs or changing system requirements). The specific value of K must be selected with care, taking into consideration the problem's characteristics and the system's constraints [81]. In this instance, since we know that k can always have a fixed value, such as 3, specializing k to 3 may be a good choice, as doing so can reduce the algorithm's complexity, resulting in a more straightforward and resource-efficient implementation.

The functions *kNN_UpdateBestCaching* and *kNN_VoteBetweenBest* are critical to this algorithm; therefore, their specialization with $k = 3$ in mind can result in significant performance gains. In The transformations applied in *kNN_UpdateBestCaching*, which can be seen in 4.3 and 4.4 , have simplified the code by eliminating the for loop used to determine the k nearest neighbors and replacing it with a series of direct comparisons specific to the case where $k = 3$. In addition, the function now returns only the distance value of the farthest point discovered, which is the second worst among the k nearest neighbors, thus eliminating the need to store and sort the k nearest neighbors.

Listing 4.3: kNN_UpdateBestCaching (v2)

```c
double kNN_UpdateBestCaching(double queryDistance, int queryIdx, int k,
                             double * bestDistances, int * bestPointsIdx)
{
    double worstOfBest = 0;
    int worstOfBestIdx = -1;
    double secondWorstOfBest = 0;
    int secondWorstOfBestIdx = -1;

    int i;
    for (i = 0; i < k; i++)
    {
        if (worstOfBest < bestDistances[i])
        {
            secondWorstOfBest = worstOfBest;
            secondWorstOfBestIdx = worstOfBestIdx;

            worstOfBest = bestDistances[i];
            worstOfBestIdx = i;
        }
        else if (secondWorstOfBest < bestDistances[i])
        {
            secondWorstOfBest = bestDistances[i];
            secondWorstOfBestIdx = i;
        }
    }

    if (queryDistance < worstOfBest)
    {
        bestDistances[worstOfBestIdx] = queryDistance;
        bestPointsIdx[worstOfBestIdx] = queryIdx;
    }
    return (queryDistance > secondWorstOfBest) ? queryDistance
                                               : secondWorstOfBest;
}
```

Listing 4.4: k3NN_UpdateBestCaching (v3)

```c
double kNN_UpdateBestCaching(double queryDistance, int queryIdx,
                             double * bestDistances, int * bestPointsIdx)
{
    if(queryDistance < bestDistances[0])
    {
        if(queryDistance < bestDistances[1])
        {
            if(queryDistance < bestDistances[2])
            {
                bestDistances[0] = bestDistances[1];
                bestPointsIdx[0] = bestPointsIdx[1];
                bestDistances[1] = bestDistances[2];
                bestPointsIdx[1] = bestPointsIdx[2];
                bestDistances[2] = queryDistance;
                bestPointsIdx[2] = queryIdx;
            }
            else
            {
                bestDistances[0] = bestDistances[1];
                bestPointsIdx[0] = bestPointsIdx[1];
                bestDistances[1] = queryDistance;
                bestPointsIdx[1] = queryIdx;
            }
            return bestDistances[0];
        }
        else
        {
            bestDistances[0] = queryDistance;
            bestPointsIdx[0] = queryIdx;
        }
    }
    return queryDistance;
}
```

In the function *kNN_VoteBetweenBest*, the code was modified by removing the histogram and loops used to count class votes. These changes can be verified in 4.5 and 4.6. Instead, the function performs a series of tests using only the three nearest points ($k = 3$) to determine the most popular class. By making $k = 3$ and getting rid of the histogram and for loops, the code can be simplified and result in more efficient hardware implementations.

Listing 4.5: kNN_VoteBetweenBest (v2)

```
CLASS_TYPE kNN_VoteBetweenBest(int *bestPointsIdx, CLASS_TYPE * training_Y,
                                int k, int n_classes)
{
    CLASS_TYPE *histogram = calloc(n_classes, sizeof(CLASS_TYPE));

    int i;

    for ( i = 0; i < k; i++)
    {
        int bestIdx = bestPointsIdx[i];
        CLASS_TYPE cl = training_Y[bestIdx];
        histogram[(int)cl]++;
    }

    CLASS_TYPE mostPopular = -1;
    int mostPopularCount = -1;

    for ( i = 0; i < n_classes; i++)
    {
        if (histogram[i] > mostPopularCount)
        {
            mostPopularCount = histogram[i];
            mostPopular = (CLASS_TYPE)i;
        }
    }

    return mostPopular;
}
```

Listing 4.6: k3NN_VoteBetweenBest(v3)

```
CLASS_TYPE kNN_VoteBetweenBest(int *bestPointsIdx, CLASS_TYPE * training_Y,
                               int n_classes)
{
    if(training_Y[bestPointsIdx[0]] == training_Y[bestPointsIdx[1]])
        return training_Y[bestPointsIdx[0]];
    else if( training_Y[bestPointsIdx[0]] == training_Y[bestPointsIdx[2]])
        return training_Y[bestPointsIdx[0]];
    else if( training_Y[bestPointsIdx[2]] == training_Y[bestPointsIdx[1]])
        return training_Y[bestPointsIdx[1]];
    else
    {
        if(training_Y[bestPointsIdx[0]] <   training_Y[bestPointsIdx[1]] &&
           training_Y[bestPointsIdx[0]] < training_Y[bestPointsIdx[2]])
            return training_Y[bestPointsIdx[0]];
        else if(training_Y[bestPointsIdx[1]] < training_Y[bestPointsIdx[0]] &&
                training_Y[bestPointsIdx[1]] < training_Y[bestPointsIdx[2]])
            return training_Y[bestPointsIdx[1]];
        else
            return training_Y[bestPointsIdx[2]];
    }
}
```

Those modifications can result in increased hardware resource utilization efficiency and decreased latency. However, specialization to $k = 3$ means that this version of the function will only operate for that particular value of k. Any other value of k will necessitate a new implementation of the function.

Version with Multiversion $k = 3$ (v4)

In certain circumstances, it may be beneficial to make use of multiple versions of the same algorithm, as is the case with the Multiversion technique. One example of this kind of scenario is when various iterations of the algorithm have significant differences in terms of both the resources they employ and the latency they experience.

Listing 4.7 show the changes done on the TOP function. These modifications seek to provide a multiversion version in which option *1* (*opt* = *1*) selects the specialized version for *k* = *3*, while option *2* (*opt*!= *1*) selects the generalised version.

In general, the specialized version for $k = 3$ can provide a significant performance advantage over a more generic version that works with any value of k. Due to the fact that *kNN* for $k = 3$ requires a different set of operations than *kNN* for other values of k, it is feasible to optimize the implementation and reduce execution time by specializing the algorithm for this specific value of k. However, the more generic version of *kNN* can be beneficial in situations where the value of k is unknown in advance, allowing for greater flexibility. By generating a version with multiversion, we enable the *HLS* system to autonomously select the best implementation based on the specified value of k. This can result in a more efficient implementation in terms of hardware resources and execution time than a single version that endeavors to handle all possible values of k. Multiple versions can, however, increase code complexity and maintenance difficulty, as well as hardware resource usage and latency.

Listing 4.7: Multiversion code (v4)

```
CLASS_TYPE kNN_Predict(DATA_TYPE *training_X, CLASS_TYPE *training_Y,
                        DATA_TYPE *queryDatapoint, DATA_TYPE *min,
                        DATA_TYPE *max, int n_features, int n_testing,
                        int n_training, int k, double *bestDistances,
                        int* bestPointsIdx, int n_classes, int opt)
{
    CLASS_TYPE voteResult;

    kNN_MinMaxNormalize(min, max, queryDatapoint, n_features);

    if(opt == 1) voteResult = knn_k3(training_X, training_Y, queryDatapoint,
                                   n_features, n_training);
    else voteResult = knn_k(bestDistances, bestPointsIdx, n_features,
                            n_testing, n_training, k, n_classes, training_X,
                            training_Y, queryDatapoint);

    return voteResult;
}
```

Version with *k*, *N_FEATURES*, *N_TRAINING*, and *N_TESTING* specialization (v5)

We saw in the previous section that specializing the variable *k* allows for optimization of the resulting circuit, allowing us to reduce the number of resources required to implement it, resulting in a smaller circuit, lower latency, and lower power consumption. Aside from specializing *k* to a fixed value of *3*, in this version we also specialize *N_FEATURES*, *N_TRAINING*, and *N_TESTING* parameters.

The number of features present in the training and testing data is represented by *N_FEATURES*. It is used to traverse the training and testing data and establish the dimension of the feature vectors. The size of the training set, i.e., the number of samples used to train the *kNN* algorithm, is represented by *N_TRAINING*. To perform classification, the algorithm computes the distance between a test instance and all training instances. *N_TESTING* is the size of the test set, which contains the samples for which the *kNN* algorithm should perform classification based on the training set's information. To identify the predicted class, the algorithm computes the distance between each test case and the training instances.

Here we consider the scenario were *N_FEATURES*, *N_TRAINING*, and *N_TESTING* have constant values *43*, *4336*, and *1082*, respectively. These specializations for the variables *N_FEATURES*, *N_TRAINING*, and *N_TESTING* do not provide a transformation as significant as the specialization of $k = 3$, as shown in the examples presented in Listing 4.8. However, this does not exclude them from contributing to performance and efficiency gains in the *kNN* algorithm. For example, knowing the exact number of features, training samples, and test samples in advance allows the compiler to perform specific optimizations, removing the need for additional logic to handle variable values, such as adjusting the sizes of vectors and matrices used in the algorithm to avoid unnecessary memory allocations and faster data access. Furthermore, the specialization of these variables enables the compiler to perform more precise static analysis, potentially resulting in optimizations such as loop unrolling or vectorization that can enhance code performance.

Listing 4.8: Code transformation example for v5

```
void kNN_PredictAll(DATA_TYPE *training_X, CLASS_TYPE *training_Y,
                    DATA_TYPE *testing_X, CLASS_TYPE *testing_Y,
                    DATA_TYPE *min, DATA_TYPE *max)
{   (...)
    for (i = 0; i < N_TESTING; i++)
    {
        testing_Y[i] = kNN_Predict(training_X, training_Y,
                                   &testing_X[i*N_FEATURES], min, max);
    }
}

CLASS_TYPE kNN_Predict(DATA_TYPE *training_X, CLASS_TYPE *training_Y,
                       DATA_TYPE *queryDatapoint, DATA_TYPE *min,
                       DATA_TYPE *max)
{   (...)

```

```
    for ( i = 0; i < N_TRAINING; i++)
    {   (...)
        for ( j = 0; j < N_FEATURES; j++)
        {
            DATA_TYPE feature = queryDatapoint[j];
            double diff = feature - training_X[i*N_FEATURES+j];
            distance += diff * diff;
        }
        (...)
    }
    (...)
}
```

In short, these variables are critical for the *kNN* algorithm's operation since they affect the feature space, the amount of data available for training and testing, and have a direct impact on distance calculation and determining the expected class for test cases. The development of a version with a specialization of them can be considered a useful study since it may contribute to the optimization of the resultant circuit by decreasing size, latency, and power consumption, improving memory utilisation, and simplifying code.

Version with Multiversion *k*, *N_FEATURES*, *N_TRAINING*, and *N_TESTING* (v6)

Similarly to Section 4.1.2, using numerous versions of the same algorithm may be advantageous by establishing a version with Multiversion, in this case generated from version *5* (*v5*).

The proposed improvements are comparable to those given to Listing 4.7.These improvements aim to give a multiversion version, with option *1* (*opt* = *1*) selecting the specialized version for *k* = *3*, *N_FEATURES* = *43*, *N_TRAINING* = *4336*, and *N_TESTING* = *1082*, and option *2* (*opt* != *1*) selecting the generalised version.

As previously stated, the customised version can provide a significant performance boost over a more generic version that can be used in any application. Because *kNN* requires a different set of operations for the variables in question than *kNN* for other circumstances, it is possible to optimize the implementation and minimise execution time by specializing the algorithm for the constant values of these variables. The generic version of *kNN*, on the other hand, can be useful in circumstances where the values of the variables are unknown at the outset, allowing for additional flexibility.

We enable the *HLS* system to generate a multiversion implementation that includes an input parameter to select on of the implementations depending on the supplied value . This can lead to a more efficient implementation in terms of hardware resources and runtime than a single version that attempts to handle all conceivable variable values. Multiple versions, on the other hand, might increase code complexity and maintenance difficulty, as well as hardware resource utilisation and delay.

Additional Versions (v7-v17)

The purpose of the additional versions is to enhance the investigation into the possible existence of a correlation between the degree of similarity between various individual versions and the number of resources required for a version using the multiversion technique for the same versions. Version *v7_fp* consists of only specializing the variables *N_FEATURES*, *N_TRAINING*, and *N_TESTING*, using the same strategy as the previous sections for the constant values of the *WI_K3_F* scenario, as presented in Section 4.1.2.

In addition, versions *8* through *17* were developed using the multiversion technique. These versions comprise all conceivable combinations between two individual versions, as well as the clusters suggested by the chosen similarity detection tool.

Vitis HLS Directives

Listing 4.9 shows pseudocode that has been labelled to show where the relevant *HLS* directives [82] have been applied. Each label in Table 4.2 is properly characterised, with the *HLS* directives used in each section of the code specified. The primary *HLS* directives utilised are loop trip-count, pipelining, loop unroll, and inlining, all of which play an important part in improving the algorithm's performance and efficiency.

Listing 4.9: Labeled kNN pseudocode)

```
Label01_UBC: double kNN_UpdateBestCaching((...))
{   (...)
    Label02_UBC_for: for (i = 0; i < k; i++) { (...) } (...)
}
Label03_IB: void kNN_InitBest((...))
{   (...)
    Label04_IB_for: for ( i = 0; i < k; i++) { (...) }
}
Label05_VBB: CLASS_TYPE kNN_VoteBetweenBest((...))
{   (...)
    Label06_VBB_for: for ( i = 0; i < k; i++) { (...) } (...)
    Label07_VBB_for: for ( i = 0; i < n_classes; i++) { (...) } (...)
}
Label08_MMN: void kNN_MinMaxNormalize((...))
{   (...)
    Label09_MMN_for: for ( i = 0; i < n_features; i++) { (...) }
}
Label10_kNN_P: CLASS_TYPE kNN_Predict((...))
{   (...)
    Label11_kNN_P_for: for ( i = 0; i < n_training; i++)
    { (...)
        Label12_kNN_P_for: for ( j = 0; j < n_features; j++) { (...) } (...)
    } (...)
}
```

Table 4.2: HLS directives applied to the kNN algorithm.

Label	TOP	Inlining	Loop Tripcount (max)	Loop Unroll	Pipelining (II)
Label01_UBC	-	yes	-	-	-
Label02_UBC_for	-	-	3	-	1
Label03_IB	-	yes	-	-	-
Label04_IB_for	-	-	3	yes	1
Label05_VBB	-	yes	-	-	-
Label06_VBB_for	-	-	3	-	1
Label07_VBB_for	-	-	6	-	1
Label08_MMN	-	yes	-	-	-
Label09_MMN_for	-	-	43	-	16
Label10_kNN_P	yes	yes	-	-	-
Label11_kNN_P_for	-	-	4336	-	-
Label12_kNN_P_for	-	-	43	-	4

The loop tripcount directive allows to provide the maximum, minimum or exact number of loop iterations, which the compiler can use to conduct static optimizations and improve code performance. The pipelining directive allows different iterations to run in parallel. This strategy improves overall performance by maximising the usage of available hardware resources and increasing system throughput. The loop unroll directive is used to expand a loop into several iterations by unrolling it. This minimises loop overhead, such as branch instructions, and allows the compiler to take use of optimization opportunities like speculative execution and data parallelism more effectively. Finally, the inlining directive instructs the compiler to replace function calls with the function's actual content, avoiding function call overhead and allowing for more efficient execution.

These *HLS* directives were applied to all algorithm versions in order to optimize performance, latency, and resource usage. By carefully employing these directives, it is feasible enhance the potential of the hardware underneath it and significantly increase the performance of the *kNN* algorithm.

4.2 Disparity

In this work, we also study the Disparity Benchmark, which is one of several benchmarks included in the San Diego Vision Benchmark Suite [71].

Comparing the images acquired by two slightly offset cameras yields stereo images for disparity estimation. The objective of this algorithm is to identify a match between the pixels of the two images and then calculate the disparity, which is a measure of the difference in the object's position between the two cameras. This can be used to calculate the scene's depth, an essential problem in computer vision applications such as autonomous vehicle navigation, object or obstacle recognition, and robotics.

Table 4.3: Disparity possible scenarios and respective input data sizes.

Scenario	**Input Data Size**
WUXGA	2.3M pixels
FULLHD	2.0M pixels (1080p format)
VGA	300k pixels
CIF	100k pixels
QCIF	25k pixels
SQCIF	12.5k pixels
SIM	Execution time around 6-10 million cycles
SIM_FAST	Execution time around 2-4 million cycles
TEST	Execution time around 10k-100k cycles

The algorithm employs sliding window cross-correlation, which compares windows in corresponding horizontal positions on the left and right images. The window with the smallest sum of absolute differences is chosen as the most likely match, and the disparity is computed as the horizontal difference between the coordinates of the window in the left image and the corresponding window in the right image. For each possible horizontal offset between the two stereo images, the algorithm is executed to determine the optimal disparity for each pixel in the left image. The output of the algorithm is an image of disparity, where the whiter pixels represent greater disparities.

Disparity is a computationally intensive benchmark and therefore presents a significant performance optimization challenge, particularly when implemented in hardware.

4.2.1 Disparity Analysis

Implemented in *C*, the disparity algorithm attempts to calculate the disparity between two stereo images using the *SAD* (Sum of Absolute Differences) method. All the different simulation scenarios used by the *Disparity* algorithm are listed in Table 4.3. This study mainly focuses on the *SIM* and *FULLHD* scenarios. Therefore, the transformations performed are adapted to the specific characteristics of these scenarios.

The algorithm's primary function is *getDisparity*, which accepts the images *Ileft* and *Iright*, the window size, *win_sz*, and the maximum shift, *max_sz*, and returns the disparity in *retDisp*. The function iterates over the possible displacements (from *0* to *max_shift*) and uses the *correlateSAD_2D* function to conduct the SAD correlation between the images. After each correlation calculation, the *findDisparity* function determines and retains the minimum disparity for each position in *retDisp*. Other auxiliary functions are also called within the function *correlateSAD_2D*, including *computeSAD*, which calculates the absolute difference between pixels of the images and stores the result in *SAD*; *padarray2* and *padarray4*, which perform edge filling in the images; *integralImage2D2D*, which calculates the integral image for *SAD*; and *finalSAD*, which calculates the final value of *SAD*.

Figure 4.1 depicts an analysis about the distribution of program execution time across functions and subroutines when scenario *FULLHD* input data are utilized, as determined by the results obtained from the *Gprof* code profiler. According to the provided data, the function *getDisparity* can be considered the top function because it appears first in the function call analysis (call graph) and accounts for *99.9%* of the time invested. The profiler also reveals that the program spent the majority of its time (*54.97%*) executing the *finalSAD* function. This indicates that this function would profit the most from code optimization and specialization. The *finalSAD* function computes the sum of absolute differences (*SAD*) for each image position. It accepts as input the image represented by the *integralImg2D2D* structure, the window size, *win_sz*, and an array for storing the *SAD* result, *retSad*. The function traverses all image positions using two nested for loops. The outer loop traverses columns, whereas the interior loop traverses rows. The *SAD* is calculated at each position using a specific formula. Some parts of the function can be targeted for specialization to enhance its performance, such as the iteration loops, which can be analysed to identify data access patterns and optimize cache memory access.

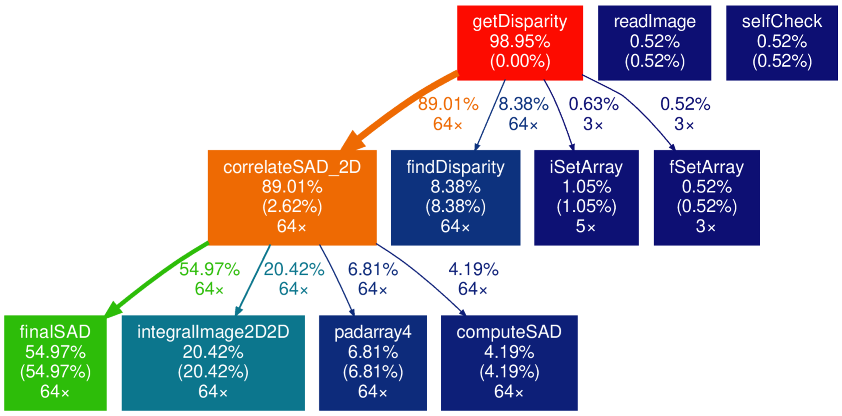

getDisparity 98.95% (0.00%)
readImage 0.52% (0.52%)
selfCheck 0.52% (0.52%)
89.01% 64×
8.38% 64×
0.63% 3×
0.52% 3×
correlateSAD_2D 89.01% (2.62%) 64×
findDisparity 8.38% (8.38%) 64×
iSetArray 1.05% (1.05%) 5×
fSetArray 0.52% (0.52%) 3×
54.97% 64×
20.42% 64×
6.81% 64×
4.19% 64×
finalSAD 54.97% (54.97%) 64×
integralImage2D2D 20.42% (20.42%) 64×
padarray4 6.81% (6.81%) 64×
computeSAD 4.19% (4.19%) 64×

Figure 4.1: Disparity Benchmark Profiling Results.

integralImage2D2D was the next most time-intensive function, consuming 20.42 % of the total time after *finalSAD*. *findDisparity* accounted for *8.38%* of the total time while *padarray4* accounted for *6.82%*. Similar to the *finalSAD* function, it is possible to optimize these functions by reducing the number of required operations, considering the structure and characteristics of the images, avoiding unnecessary copies and index manipulations, improving data access, and decreasing read/write latency in memory. The remaining functions enumerated in the profiler output required less time. The functions *correlateSAD_2D*, *iSetArray*, and *computeSAD* accounted for *2.62%*, *1.05%*, and 4.19 percent, respectively, of the total time. The remaining functions accounted for less than *1%* of the total time. Typically, the disparity algorithm requires intensive computations and operations on data arrays. Utilising monitoring tools such as "Value Counter Monitor," as described in Section 3.5 of Chapter 3, would be advantageous. Through the monitoring tool "*Value Counter Monitor*," it was possible to conduct a more in-depth analysis of certain algorithmic variables and intermediate values. It was determined that there are variables with constant values in the *SIM* scenario. The *win_sz* variable, *rows*, and *cols* with respective constant values of

4, *58*, and *76* stand out. These variables are responsible for a significant number of operations and iterations in the disparity algorithm, so they play a crucial role. In addition to the variables mentioned previously, it was observed that some intermediate values of the algorithm are frequently allocated the value zero, as is the case in the cited article [75]. Figure 4.2 displays the histogram of the diff variable of the *computeSAD* function, which was provided by the monitoring utility. These variables will be examined in greater depth in Section 4.2.2. This type of information is crucial because it permits the specialization of these values, thereby simplifying and avoiding unnecessary operations, optimizing the code, making the processing more efficient and faster, reducing the computational load, and enhancing the algorithm's overall performance. In addition, it is essential to emphasise that possible specializations should always be evaluated in relation to the problem's requirements and the characteristics of the data.

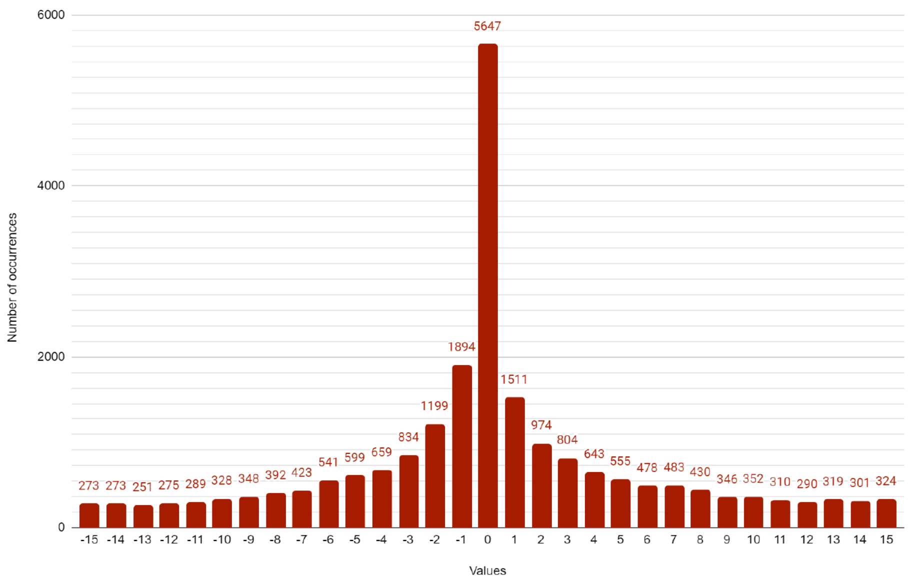


Figure 4.2: Histogram of variable "diff", from computeSAD function, possible values and number of ocurrences.

4.2.2 Disparity Optimizations

In this section, several versions of the *Disparity* algorithm resulting from transformations aimed at specialization and optimization of the code for execution on *FPGAs* using *HLS* tools will be analysed. The main distinctions between each version in terms of optimization strategies will be highlighted. How each version attempts to improve performance by reducing the number of latency cycles, decreasing resource consumption and improving the use of specific functions will be discussed.

Generic version (v1)

In the original version of the algorithm *disparity* used, only minor changes were made to make the code compatible and suitable for the version of the *vitis HLS* tool used. Some limitations of the tool regarding certain types of code, which led to these changes, are the use of features like pointers to functions and dynamic memory allocation. It is important to consult the official documentation for detailed information about supported features and possible limitations of the tool.

All subsequent versions of the *disparity* algorithm have been derived from the generalised version. This strategy allows the exploration of various configurations and the adjustment of parameters for each individual version. Since the base version was adaptive, it was possible to implement transformations and optimizations with greater freedom, adjusting the algorithm to various contexts and individual needs. This concept of producing versions derived from the generalised version allows a systematic and comparative approach to evaluating the performance and effectiveness of each modification of the algorithm.

Specialized version for *win_sz = 4* (v2)

The *win_sz* parameter specifies the size of the window utilised during disparity matching. This window determines the number of pixels compared between stereo images for disparity calculation. The significance of this parameter is dependent on the algorithm's precision and efficacy. A larger value for *win_sz* enables more precise matching, but at the expense of an increase in computational complexity. A lesser value, on the other hand, may result in a quicker but potentially less accurate match. Therefore, the appropriate selection of the *win_sz* parameter depends on the desired trade-off between precision and efficacy in the context of the particular application.

Significant benefits can be gained by setting the *win_sz* parameter to a constant value and taking advantage of its particular characteristics. By having a fixed value for *win_sz*, it is possible to eliminate the logic of manipulating this variable parameter and allocate only what is required for this fixed size, resulting in simpler code and less use of *FPGA* resources, such as memory or processing units. It is possible to implement prepossessing or caching strategies that are optimal for that particular value. These techniques can be used to enhance data access during algorithm execution, resulting in a performance boost. For instance, one can perform computations in advance or cache pertinent information so that the required data is readily accessible when the algorithm requires it. This reduces data access latency and accelerates processing as a whole, enhancing the algorithm's performance.

However, specialization in a particular value of *win_sz* may reduce the circuit's flexibility and generalizability. The algorithm may not be appropriate for images with distinct characteristics, limiting its applicability in other contexts or situations. When operating with a fixed value of *win_sz*, sufficient memory must be allocated to store the window of that size. In some instances, this can lead to redundant memory consumption if the algorithm is executed with smaller contexts. The specific value of *win_sz* should be chosen with caution, taking into consideration the problem's characteristics and the system's limitations.

We previously saw in Section 4.2.1, with the help of the monitoring tool, that for the *SIM* scenario, this parameter is always assigned a constant value of *4*.

The objective of this version of the algorithm is to evaluate the results after the *win_sz* parameter has been set to a constant value of *4*. In order to accomplish this, the parameter was initially defined as the constant *WIN_SZ* with the value *4*, allowing the elimination of the *win_sz* variable argument in all functions that utilise it and the simplification of certain operations, as demonstrated in Listing 4.11. In addition, the specialization of the *win_sz* variable primarily benefits loops that traverse the dimensions of matrices, such as in Listings 4.10 and 4.11, because the loop is executed more efficiently and optimally for the constant value of *win_sz*.

By implementing these modifications, it is possible to optimize the code by exploiting the properties and structures of the data, thereby reducing the number of operations and enhancing the algorithm's performance. These modifications can also improve the utilisation of hardware resources and reduce latency. Note, however, that specialization to a fixed value of *win_sz*, such as *win_sz = 4*, makes the function valid only for that specific value, necessitating a new implementation if a different value is required.

Listing 4.10: Disparity code win_sz transformations: original version)

```
void getDisparity(I2D* Ileft, I2D* Iright, int win_sz, int max_shift, (...))
{   (...)
    int half_win_sz, rows, cols; (...)
    half_win_sz = win_sz/2; (...)

    if(win_sz > 1)
    { (...) }
    else
    { (...) }  (...)
    retSAD.height = rows-win_sz;
    retSAD.width = cols-win_sz; (...)

    for( k=0; k<max_shift; k++)
    {
        correlateSAD_2D(&IleftPadded, &IrightPadded, &Iright_moved, win_sz, k,
        &SAD, &integralImg, auxRS); (...)
    }
}

void finalSAD(F2D* integralImg, int win_sz, F2D* retSAD)
{   (...)
    for(j=0; j<(endC-win_sz); j++) { for(i=0; i<(endR-win_sz); i++) { (...) }}
    (...)
}
```

Listing 4.11: Disparity code win_sz transformations: version 2)

```
void getDisparity(I2D* Ileft, I2D* Iright, int max_shift, I2D* retDisp)
{   (...)
```

```
    int half_win_sz, rows, cols; (...)
    half_win_sz = 2; (...)

    retSAD.height = rows-WIN_SZ_4;
    retSAD.width = cols-WIN_SZ_4; (...)

    for( k=0; k<max_shift; k++)
    {
        correlateSAD_2D(&IleftPadded, &IrightPadded, &Iright_moved, k,
        &SAD, &integralImg, auxRS); (...)
    }
}

void finalSAD(F2D* integralImg, F2D* retSAD)
{   (...)
    for(j=0; j<(endC-WIN_SZ_4); j++){for(i=0; i<(endR-WIN_SZ_4); i++){ (...) }}
    (...)
}
```

Multiversion version for *win_sz = 4* (v3)

Multiple versions of the same algorithm, as is the case with the multiversion technique, may be advantageous under specific conditions.

In this version, an attempt is made to provide a two-version version, where the option *1* (*opt = 1*) selects the specialized version for *win_sz = 4* and the option *2* (*opt != 1*) selects the generalised version. The modifications applied to the TOP function *getDisparity* are displayed in Listing 4.12. These modifications aim to provide a multiversion version in which the option *1* (*opt = 1*) selects the specialized version for win_sz = 4 and the option *2* (*opt != 1*) option selects the generalised version.

Listing 4.12: Disparity code Multiversion (v3))

```
void getDisparity(I2D* Ileft, I2D* Iright, int win_sz, (...), int opt)
{   (...)
    if(opt == 1)
    {
        retSAD.height = rows-WIN_SZ_4;
        retSAD.width = cols-WIN_SZ_4;
        for( k=0; k<max_shift; k++)
        {
            correlateSAD_2D_MV(&IleftPadded, &IrightPadded, &Iright_moved, k,
            &SAD, &integralImg, auxRS); (...)
        }
    }
    else
    {
        retSAD.height = rows-win_sz;
        retSAD.width = cols-win_sz;
        for( k=0; k<max_shift; k++)
        {
            correlateSAD_2D(&IleftPadded , &IrightPadded , &Iright_moved , win_sz ,
            k, &SAD, &integralImg , auxRS); (...)
        }
    }
}
```

As previously mentioned, producing a version with multiversion may enable the *HLS* system to select the optimal implementation based on the specified value of the specific parameter. This could result in a more efficient implementation in terms of hardware resources and execution time than one version that attempts to accommodate all possible parameter values. Multiple versions can however increase code complexity and maintainability, as well as hardware resource consumption and latency.

Zero-utilisation versions (v4-v8)

The disparity algorithm involves a series of intensive calculations and operations on data matrices, which, as we saw in Section 4.2.1 through the monitoring tool *Value Counter Monitor*, involve parameters, variables, or intermediate values to which values are assigned with a higher frequency than others and even frequently constant. It was determined that the majority of these parameters are typically designated the value zero in the *SIM* scenario. The purpose of the *v4-v8* versions is to investigate the effect of parameter specialization in an effort to optimize the algorithm.

As most of the parameters in concern are assigned their values at runtime, setting them to constant values may compromise the algorithm's functionality. In the aforementioned versions, we will therefore employ the multiversion technique, in which there will always be a generic option that conforms to the different possible values of the parameters. This allows us to guarantee the algorithm's adaptability and proper operation regardless of the values of its parameters. It was decided to investigate separately the impact of specializing these parameters in each function that uses them, resulting in versions *v4–v7*. Version *v4* specializes the *brows* variable in the function *padarray4*, which the monitoring tool revealed to have a constant value of zero. Its restriction to a constant value of zero facilitated the simplification of an operation, as shown in Listings 4.13.

Listing 4.13: Disparity version 4 code transformation: padarray4)

```
ORIGINAL:
void padarray4(I2D* inMat, I2D* borderMat, int dir, I2D* paddedArray)
{    (...)
    bRows = borderMat->data[0]; (...)
    newRows = rows + bRows; (...)

    if(dir == 1) { (...) }
    else
    {    for(i=0; i<rows-bRows; i++)
            for(j=0; j<cols-bCols; j++)
                subsref(paddedArray,(bRows+i),(bCols+j)) = subsref(inMat,i,j);
    } (...)
}

VERSION 4:
void padarray4(I2D* inMat, I2D* borderMat, int dir, I2D* paddedArray)
{   (...)
    bRows = borderMat->data[0]; (...)
    newRows = rows; (...)

    if(dir == 1) { (...) }
    else
    {   for(i=0; i < rows; i++)
            for(j=0; j<cols-bCols; j++)
                subsref(paddedArray,i, (bCols+j)) = subsref(inMat,i,j);
    } (...)
}
```

In the function *finalSAD*, it was determined that the majority of intermediate values that were part of the operation that resulted in the variable *subsref(retSAD,i,j)* were designated the value zero the majority of the time during execution. As can be seen in Listing 4.14, in version *5* these parameters are again specialized to a constant value of *0*, allowing the algorithm to be simplified.

Listing 4.14: Disparity version 5 code transformation: finalSAD)

```
ORIGINAL:
void finalSAD(F2D* integralImg, int win_sz, F2D* retSAD)
{   (...)
    for(j=0; j<(endC-win_sz); j++)
        for(i=0; i<(endR-win_sz); i++) subsref(retSAD,i,j) = subsref(
        integralImg,(win_sz+i),(j+win_sz))+subsref(integralImg,(i+1),(j+1))-
        subsref(integralImg,(i+1),(j+win_sz))-subsref(integralImg,(win_sz+i),
        (j+1)); (...)
}

VERSION 5:
void finalSAD(F2D* integralImg, int win_sz, F2D* retSAD)
{   (...)
    for(j=0; j<(endC-win_sz); j++)
    {   for(i=0; i<(endR-win_sz); i++)
        {   if(subsref(integralImg,(i+1),(j+1)) == 0 && subsref(integralImg,
            (i+1),(j+win_sz)) == 0 && subsref(integralImg,(win_sz+i),(j+1)) ==
            0) subsref(retSAD,i,j) = subsref(integralImg,(win_sz+i),(j+win_sz));
            else subsref(retSAD,i,j) = subsref(integralImg,(win_sz+i),
            (j+win_sz))+subsref(integralImg,(i+1),(j+1))-subsref(integralImg,
            (i+1),(j+win_sz)) - subsref(integralImg,(win_sz+i),(j+1));
        }
    } (...)
}
```

Analysing the results acquired using the variable monitoring tool, we notice in Figure 4.2 that the *diff* variable has values assigned to it most of the time as *0, 1, 2, -1, and -2*. Based on this analysis, additional conditions were added to this modified version in order to handle the common *diff* values, as shown in Listing 4.15.

Listing 4.15: Disparity version 6 code transformation: computeSAD)

```
ORIGINAL:
void computeSAD(I2D *Ileft, I2D* Iright_moved, F2D* SAD)
{    (...)
     for(i=0; i<rows; i++)
     {    for(j=0; j<cols; j++)
          {    diff = subsref(Ileft,i,j) - subsref(Iright_moved,i,j);
               subsref(SAD,i,j) = diff * diff;
          }
     } (...)
}

VERSION 6:
void computeSAD(I2D *Ileft, I2D* Iright_moved, F2D* SAD)
{    (...)
     for(i=0; i<rows; i++)
     {    for(j=0; j<cols; j++)
          {    diff = subsref(Ileft,i,j) - subsref(Iright_moved,i,j);
               if(diff == 0) subsref(SAD,i,j) = 0;
               else if(diff == 1 || diff == -1) subsref(SAD,i,j) = 1;
               //else if(diff == 2 || diff == -2) subsref(SAD,i,j) = 4;
               else subsref(SAD,i,j) = diff * diff;
          }
     } (...)
}
```

These modifications seek to, again, take advantage of the frequent value patterns in *diff* and simplify operations, thereby reducing the number of required calculations. By removing redundant operations and simplifying the conditions, we can reduce the algorithm's computational load and enhance its performance. In terms of efficacy and efficiency, simplifying a multiplication-based operation, such as the *computeSAD* function, can have significant effects. In general, multiplications require more computational resources than additions and subtractions. We can avoid unnecessary multiplications by specializing the function and simplifying the *SAD* calculation based on the common diff values. By reducing the number of multiplications, we decrease the consumption of hardware resources, lower the total computational load, and as a result, enhance execution time. In version *7*, the *integralImage2D2D* function performs the transformations using a process analogous to that of version *5*, as shown in Listing 4.16.

Listing 4.16: Disparity version 7 code transformation: IntegralImage2D2D)

```
ORIGINAL:
void integralImage2D2D(F2D* SAD, F2D* integralImg)
```

```
{    (...)
     for(i=1; i<nr; i++) for(j=0; j<nc; j++) subsref(integralImg,i,j) =
     subsref(integralImg,(i-1),j) + subsref(SAD,i,j);

     for(i=0; i<nr; i++) for(j=1; j<nc; j++) subsref(integralImg,i,j) =
     subsref(integralImg,i,(j-1)) + subsref(integralImg,i,j); (...)
}

VERSION 7:
void integralImage2D2D(F2D* SAD, F2D* integralImg)
{    (...)
     for(i=1; i < nr; i++) for(j=0; j < nc; j++)
     {    if(subsref(integralImg,(i-1),j) == 0 && subsref(SAD,i,j) == 0 )
              subsref(integralImg,i,j) = 0;
          else subsref(integralImg,i,j) = subsref(integralImg,(i-1),j) +
          subsref(SAD,i,j);
     }

     for(i=0; i < nr; i++) for(j=1; j < nc; j++)
     {    if(subsref(integralImg,i,(j-1)) == 0 && subsref(integralImg,i,j) == 0)
              subsref(integralImg,i,j) = 0;
          else subsref(integralImg,i,j) = subsref(integralImg,i,(j-1)) +
          subsref(integralImg,i,j);
     } (...)
}
```

Version *8* incorporates all previous versions.

By eliminating superfluous operations and simplifying the conditions, it should be possible to reduce the algorithm's computational burden and enhance its performance. In addition, this strategy may result in a more efficient use of hardware resources, which may lead to increased efficiency and decreased latency.

Specialized version for constant number of *rows* and *cols* (v9)

The variables *rows* and *cols* are essential to the disparity algorithm, as they represent the dimensions of the data matrix on which the algorithm is operating and are required to correctly iterate over the matrix elements and perform calculations and operations. The monitoring tool, in Section 4.2.1, revealed that these variables are assigned constant values of *58* and *76*, respectively, in the *SIM* scenario. Version *9* seeks to study the effects of specialization of the *rows* and *cols* variables for these constant values. In order to accomplish this, the *ROWS* and *COLS* parameters were initially defined as constants, allowing the elimination of the *rows* and *cols* variable arguments in all functions that use them and the simplification of certain operations, as shown in Listings 4.17 and 4.18.

Listing 4.17: Disparity code transformations example: original)

```
void correlateSAD_2D(I2D*Ileft,I2D*Iright,I2D* Iright_moved,int win_sz,(...))
```

```
{    int rows, cols; (...)

     rows = Iright_moved->height;
     cols = Iright_moved->width;

     for(i=0; i < rows*cols; i++) asubsref(Iright_moved,i) = 0; (...)
}
void computeSAD(I2D *Ileft, I2D* Iright_moved, F2D* SAD)
{    int rows, cols, i, j, diff;

     rows = Ileft->height;
     cols = Ileft->width;

     for(i=0; i<rows; i++)
     {    for(j=0; j<cols; j++){ (...) }} (...)
}
```

As was the case with the specialization of the *win_sz* parameter, one advantage of the specialization of these variables to a constant value is the ability to apply specific optimization techniques to this fixed size. For instance, it is feasible to perform preprocessing or optimize cache settings for this particular capacity, resulting in enhanced performance. In addition, specialization can simplify code and reduce the intricacy of iterations, resulting in decreased resource consumption and a more effective implementation. A significant disadvantage of specialization is the restriction of the algorithm to a single matrix size. specialization restricts the algorithm's adaptability, making it less generic and applicable to various application scenarios.

Listing 4.18: Disparity code transformations example: v9)

```
void correlateSAD_2D(I2D*Ileft,I2D*Iright, I2D*Iright_moved,int win_sz,(...))
{    (...) // prodRowsCols = rows * cols
     for(i=0; i< prodRowsCols ; i++) asubsref(Iright_moved,i) = 0; (...)
}
void computeSAD(I2D *Ileft, I2D* Iright_moved, F2D* SAD)
{    (...)
     for(i=0; i<ROWS; i++)
     {    for(j=0; j<COLS; j++) { (...) }
     } (...)
}
```

By specializing the rows and columns variables to a constant value, it is possible to obtain optimization and performance benefits, but at the expense of code flexibility and reusability in various contexts. The decision regarding whether or not to specialize these variables should be based on the specific requirements of the undertaking at hand.

Multiversion version for version 9 (v10)

In this version, as in previous versions, an attempt is made to provide a two-version version in which option *1* (*opt* = *1*) selects the specialized version *9*, in this case for *rows* = *58* and *cols* =

76, and option *2* (*opt!= 1*) selects the generalised version. Similar modifications are required as those in Listing 4.12.

As previously mentioned, producing a version with multiversion may enable the *HLS* system to select the optimal implementation based on the specified value of the relevant parameter. This could result in a more efficient implementation in terms of hardware resources and execution time than a single variant that attempts to accommodate all possible parameter values. Multiple versions can, however, increase code complexity and maintainability, as well as hardware resource consumption and latency.

Specialized version for constant number of *rows, cols* and *win_sz* (v11)

In the preceding sections, we observed that specializing for constant values the parameters *win_sz*, *rows* and *cols* separately makes possible an overall improvement in code performance, optimization of the resulting circuit, and a reduction in the amount of resources required to implement it, resulting in a smaller circuit, lower latency, and lower power consumption. The purpose of this version is to analyze the situation where the values of the three parameters are specialized to constants.

Multiversion version for version 11 (v12)

In this version, as in previous versions, an attempt is made to give a two-version version, with option *1* (*opt = 1*) selecting the specialized version *11*, in this case for *win_sz = 4*, *rows = 58*, and *cols = 76*, and option *2* (*opt!= 1*) selecting the generalist version. The modifications required are identical to those shown in Listing 4.12.

Additional Versions (v13-v17)

The objective of the additional versions is to further our investigation towards the possible existence of a correlation between the degree of similarity between various individual versions and the number of resources required for a version that employs the multiversion technique for the same versions. Versions *13* through *17* were created using the multiversion technique. These versions cover all possible combinations between two individual versions, as well as the clusters suggested by the chosen similarity detection tool.

Vitis HLS directives

Listing 4.19 displays pseudocode that has been labeled to indicate where the required *HLS* directives [82] have been applied. Each label in Table 4.4 is properly described, with the *HLS* directives applied in each portion of the code. The major *HLS* directives utilized are loop tripcount, pipelining, and inlining, all of which help to improve the algorithm's performance and efficiency. Section 4.1.2 has already characterised these directives.

Listing 4.19: Labeled Disparity pseudocode)

```
Label01_gD: void getDisparity((...))
{   (...)
    Label02_gD_for: for(i=0; i<nr; i++) {
        Label03_gD_for: for(j=0; j<nc; j++) {(...)}} (...)
    Label04_gD_for: for(i=0; i<nr; i++) {
       Label05_gD_for: for(j=0; j<nc; j++) {(...)}} (...)
    Label06_gD_for: for(i=0; i<1; i++) {
       Label07_gD_for: for(j=0; j<2; j++) {(...)}} (...)

    if(win_sz > 1) { (...) }
    else{(...)
        Label08_gD_for:for(i=0; i<IleftPadded.height; i++) {
            Label09_gD_for:for(j=0; j<IleftPadded.width; j++) {(...)}} (...)
        Label10_gD_for:for(i=0; i<IrightPadded.height; i++) {
            Label11_gD_for:for(j=0; j<IrightPadded.width; j++) {(...)}}}(...)
        Label12_gD_for:for(i=0; i<rows; i++) {
            Label13_gD_for:for(j=0; j<cols;j++) {(...)}
    }(...)
    Label14_gD_for: for(i=0; i<rows; i++) {
       Label15_gD_for: for(j=0; j<cols; j++) {(...)}} (...)
    Label16_gD_for: for(i=0; i<rows; i++) {
       Label17_gD_for: for(j=0; j<cols; j++) {(...)}} (...)
    Label18_gD_for:for( k=0; k<max_shift; k++) {(...)}
}

Label01_crSAD: void correlateSAD_2D((...)){(...)
    Label02_crSAD_for:for(i=0;i<rows*cols;i++) (...)}

Label01_II2D2D: void integralImage2D2D(F2D* SAD, F2D* integralImg)
{   (...)
    Label02_II2D2D_for:for(i=0; i<nc; i++) (...)

    Label03_II2D2D_for:for(i=1; i<nr; i++)
        Label04_II2D2D_for:for(j=0; j<nc; j++) {(...)}
    Label05_II2D2D_for:for(i=0; i<nr; i++)
        Label06_II2D2D_for:for(j=1; j<nc; j++) (...)
}

Label01_p2: void padarray2((...))
{   (...)
    Label02_p2_for:for(i=0; i<newRows; i++) {
        Label03_p2_for: for(j=0; j<newCols; j++) {(...)}}
    Label04_p2_for:for(i=0; i<rows; i++)
        Label05_p2_for:for(j=0; j<cols; j++)(...)
}

Label01_p4:void padarray4((...))
{   (...)
    if(dir ==1)
    {   Label02_p4_for:for(i=0; i<rows; i++)
            Label03_p4_for:for(j=0; j<cols; j++)(...)
    }else {
        Label04_p4_for:for(i=0; i<rows-bRows; i++)
            Label05_p4_for:for(j=0; j<cols-bCols; j++) (...)
    }(...)
}

Label01_cSAD:void computeSAD((...))
{   (...)
    Label02_cSAD_for:for(i=0; i<rows; i++)
    {   Label03_cSAD_for:for(j=0; j<cols; j++) {(...)}} (...)
}

Label01_fSAD:void finalSAD((...))
{   (...)
    Label02_fSAD_for:for(j=0; j<(endC-win_sz); j++)
    {   Label03_fSAD_for:for(i=0; i<(endR-win_sz); i++){(...)}} (...)
}

Label01_fD:void findDisparity((...))
{   (...)
    Label02_fD_for:for(i=0; i<nr; i++)
    {   Label03_fD:for(j=0; j<nc; j++){(...)}}(...)
}
```

The *HLS* directives have been implemented to all versions of the algorithm to improve performance, latency, and resource use. It is feasible to maximize the capability of the underlying hardware and considerably improve the performance of the algorithm disparity by properly implementing these directives.

4.3 Summary

This chapter conducts a study of the benchmarks adopted for the development of this work. The analysis of benchmarks allows the identification and optimization of crucial code regions, resulting in solutions that are more effective and scalable. For each benchmark, algorithm features and profiling results are provided. In consideration of this initial analysis, the potential transformations that will give allow the development of the various versions of each algorithm are presented and justified, as are the *Vitis HLS* tool directives that were applied to each of them. It is discussed how each version attempts to improve performance by decreasing the number of latency cycles, decreasing resource consumption, and enhancing the utilisation of particular functions.

kNN requires distance calculations and result ordering, which can be computationally intensive, making it a suitable benchmark for this investigation. Its simplicity makes it simple to analyse, without the need for profiling tools, and enables comparisons between various code optimization strategies. Its *TOP* function is *knn_predict* due to its computationally intensive characteris-

Table 4.4: HLS directives applied to the Disparity algorithm.

Label	TOP	Inlining	Loop Tripcount (max)	Pipelining (II)	Label	TOP	Inlining	Loop Tripcount (max)	Pipelining (II)
Label01_gD	yes	yes	-	-	**Label05_II2D2D_for**	-	-	58	6
Label02_gD_for	-	-	54	1	**Label06_II2D2D_for**	-	-	76	6
Label03_gD_for	-	-	72	1	**Label01_p2**	-	yes	-	-
Label04_gD_for	-	-	54	1	**Label02_p2_for**	-	-	58	1
Label05_gD_for	-	-	72	1	**Label03_p2_for**	-	-	76	1
Label06_gD_for	-	-	54	1	**Label04_p2_for**	-	-	54	1
Label07_gD_for	-	-	72	1	**Label05_p2_for**	-	-	72	1
Label08_gD_for	-	-	54	1	**Label01_p4**	-	yes	-	-
Label09_gD_for	-	-	72	1	**Label02_p4_for**	-	-	58	1
Label10_gD_for	-	-	58	1	**Label03_p4_for**	-	-	76	1
Label11_gD_for	-	-	76	1	**Label04_p4_for**	-	-	58	1
Label12_gD_for	-	-	58	1	**Label05_p4_for**	-	-	76	1
Label13_gD_for	-	-	76	1	**Label01_cSAD**	-	yes	-	-
Label14_gD_for	-	-	58	1	**Label02_cSAD_for**	-	-	58	1
Label15_gD_for	-	-	76	1	**Label03_cSAD_for**	-	-	76	1
Label16_gD_for	-	-	8	1	**Label01_fSAD**	-	yes	-	-
Label01_crSAD	-	yes	-	-	**Label02_fSAD_for**	-	-	72	2
Label02_crSAD_for	-	-	4408	1	**Label03_fSAD_for**	-	-	54	2
Label01_II2D2D	-	yes	-	-	**Label01_fD**	-	yes	-	-
Label02_II2D2D_for	-	-	76	1	**Label02_fD_for**	-	-	54	10
Label03_II2D2D_for	-	-	58	6	**Label02_fD_for**	-	-	72	10
Label04_II2D2D_for	-	-	76	6	**-**	-	-	-	-

tics. Table 4.1.2 lists the various simulation scenarios than can be used in this *kNN* algorithm, but the transformations conducted are specific to the particular characteristics of the *WI_K3_F* and *WI_K20_F* scenarios. It was necessary to develop a more generalised version of this algorithm for use with the *HLS* tool, as the initial version was already highly specialized. Thus, we have a sufficient foundation for generating additional versions and comparing them. specializations of parameters such as *k*, *N_FEATURES*, *N_TRAINING* and *N_TESTING* are the starting point of the transformations that generate the remaining versions. The parameter *k* is given special consideration due to the fact that, for the scenario in question, it can always have a fixed value of *3*, which is a very small value in comparison to the other parameters' potential specializations. A specialization of $k = 3$ can originate a substantial amount of transformations in the essential functions like *kNN_UpdateBestCaching* and *kNN_VoteBetweenBest*, which may result in substantial performance improvements. Taking into consideration the specialized versions, the remaining versions are the result of the application of the multiversion technique, some of which were suggested by the code similarity analysis tool described in Section 3.6, Chapter 3. Loop tripcount, pipelining, loop unrolling, and inlining are the primary directives used by *HLS* tool to enhance the efficacy and efficiency of this algorithm in its various implementations.

The primary objective of the Disparity algorithm is to identify a match between the pixels of the two images and then calculate disparity, a measure of the difference in object position between the two cameras. It is a computationally intensive algorithm, providing a significant challenge for performance optimization, especially when implemented in hardware. Table 4.2.1 lists the various simulation scenarios utilised by the disparity algorithm. The conducted transformations are modified according to the particular characteristics of the *SIM* and *FULLHD* scenarios. Profiling these programs required the use of the *Gprof* and *G2prof* tools due to their increased size and complexity. According to the data provided by the tools, the function *getDisparity* is the most important as it occupies the first position in the function call analysis (call graph) and is responsible for *99.9%* of the time invested. The profiler also reveals that the function *finalSAD* consumed the second majority of the program's time (*54.97%*). This indicates that code optimization and specialization would also be most advantageous for this function. Some portions of the function could be targeted for specialization in order to improve its performance by reducing the number of required operations, taking into account the structure and characteristics of the images, avoiding unnecessary copies and index manipulations, enhancing data access, and reducing read/write latency in memory.

To make the code compatible and suitable for the version of the *vitis HLS* tool used, only minimal modifications were made to the original version of the disparity algorithm. From this version, all succeeding versions of the disparity algorithm were derived. Since the base version was adaptive, it was possible to implement transformations and optimizations with greater flexibility, adapting the algorithm to various contexts and individual requirements while also permitting a systematic and comparative evaluation of the performance and efficacy of each algorithm modification. As the disparity algorithm requires intensive calculations and operations on data matrices, monitoring instruments such as the "*Value Counter Monitor*" described in Section 3.6 of Chapter 3 were found advantageous. Using this monitoring tool, it was feasible to conduct a more comprehensive analysis of particular algorithmic variables and intermediate values. The variables *win_sz*, *rows*, and *cols* are distinguished by their constant values of *4*, *58*, and *76*, respectively, in *SIM* scenario. In the disparity algorithm, these variables are responsible for a significant number of operations and iterations, so they played a crucial role in the developing of some specialized versions. In addition, it was observed that some intermediate values of the algorithm are frequently allocated the value zero, as the example described in the article [75]. From this concept, zero-based versions were developed. This type of information is vital because it permits the specialization of these values, thereby simplifying and avoiding unnecessary operations, optimizing the code, making the processing more efficient and faster, reducing the computational burden, and enhancing the algorithm's overall performance. Taking into account the specialized versions, the remaining versions are the result of the multiversion technique, some of which were suggested by the code similarity analysis tool described in Section 3.6, Chapter 3. As mentioned previously, generating a version with multiversion may enable the *HLS* system to select the optimal implementation based on the value of the specified parameter. This could lead to a more efficient implementation in terms of hardware resources and execution time than a version that attempts to accommodate all possible parameter values. Multiple versions can, however, increase the code's complexity and maintainability, as well as hardware resource consumption and latency. The *HLS* directives used were loop tripcount, pipelining, and inlining. They play a significant role in enhancing the efficacy and efficiency of the algorithm.

In general, the selected benchmarks for our analysis exhibit substantial room for enhancement and provide useful diversity in their structures and modes of operation. It is crucial to emphasise that potential specializations should always be evaluated with regard to of the problem's requirements and the characteristics of the data.

Chapter 5

Experimental Results

This chapter's main goal is to describe the experimental setup and expose, analyse, and justify the results acquired by the proposed approach when applied to the benchmarks described in Section 3.8 of Chapter 3 and in Chapter 4.

Section 5.1 describes the setup for the experiment used to generate the subsequent sections' results. The purpose of this description is to ensure the reproducibility and openness of the experiments. The obtained results for the *pow*, *kNN*, and *Disparity* benchmarks are presented, analysed, and justified in Section 5.2. The results are organised by pertinent metrics such as execution time, resource utilisation (*FFs*, *BRAMs*, *DSPs*, and *LUTs*), and area-delay product (*ADP*) and presented in tables. These metrics provide a comprehensive view of the performance, efficiency, and resource usage of approach-optimized versions. Section 5.3 presents the results of the study presented in Section 3.6 of Chapter 3 about the relationship between the resource usage resulting from multiversion versions and the degree of similarity of the versions that compose them. Section 5.4 presents the results of the extension made to the "monitors" library, also referred in Section 3.5 of Chapter 3.

Section 5.6 concludes with an evaluation of the obtained results and a comparison with those reported in the state-of-the-art. We include a discussion of the principal conclusions and findings regarding the benefits and limitations of the proposed approach. We a focus on the performance enhancements achieved, the resource utilisation efficiency, and potential improvement areas for future research.

5.1 Experimental Setup

This research was conducted on a *LAPTOP-HTC0Q8HI* equipped with an *Intel(R) Core(TM) i7-8750H* processor and *16 GB* of *RAM*. Targeting the *Zynq-7000 SoC XC7Z020-1CLG400C FPGA*, optimized versions of the benchmarks were simulated and synthesised using *Xilinx's Vitis HLS 2022.2* tool. The simulation and synthesis reports provided essential information, which includes the estimated latency, the number of clock cycles required for correct operation, the maximum frequency, and the estimated usage of resources such as *BRAMs*, *LUTs*, *DSPs*, and *FFs*. The total

Table 5.1: Total available resources of the Zynq-7000 SoC XC7Z020-1CLG400C FPGA, present in the PYNQ™-Z2 board.

LUTs	FFs	RAM (36 Kb)	DSP (25x18 mux)	PLLs	Programmable input/output pins	Total Resources
85000	53200	560	220	5	200	139185

available resources of the Zynq-7000 SoC XC7Z020-1CLG400C FPGA, present in the PYNQ™-Z2 board are presented in Table 5.1.

The results obtained are used to compare each optimized version to its associated generic or original version, using metrics such as speedup, percentage of resources used, and area-delay product. Speedup indicates the degree to which the optimized variant outperforms its generic or original equivalent. The percentage of resources utilised represents the efficiency with which available hardware resources are utilised in comparison to the respective original or generic versions. The area-delay product is a metric that takes into consideration both the implementation's area (resources) and latency.

These metrics and analysis tools are essential for evaluating the performance, efficiency, and viability of the approach's optimized versions. They permit an objective and comparative analysis, which facilitates the identification of significant enhancements.

5.2 HLS Synthesis Results

This section presents and analyses the results obtained by applying the proposed approach to the selected benchmarks. These benchmarks include the motivating example, the *pow* function, the *kNN* algorithm implementation from the *SPEC* group, as well as Disparity from the *SD-VBS* group.

The optimized versions of each benchmark are compared to the original or generic versions of the same, taking into account metrics like execution time, latency cycles, resource utilisation, and area-delay product. We consider the modifications made by the optimized versions and examine their impact on performance, including the reduction of latency cycles and the more effective use of hardware resources.

Through this comprehensive analysis of the results, we hope to provide a clear and substantiated view of the performance and benefits obtained by the proposed approach, enabling a proper evaluation of its efficacy in the optimization of selected benchmarks.

5.2.1 *Pow* function

The results of the synthesis regarding the latency cycles and number of resources utilised by each version of the *pow* function are presented in Tables 5.2 and 5.4, respectively. In Tables 5.3 and 5.5, the comparison of speedups and resource usage percentages relative to the generic version *v1* can also be verified. Section 3.8 of Chapter 3 previously presented this program and its respective versions.

Table 5.2: Results after pow versions simulations of the values referring to the execution time with a target of 10ns and uncertainty of 2.70 ns. (single-precision float (SPF); maximum frequency (Fmax); multiversion (MV); specialized (SPEC))

Version	**Functionality**	**Estimated (ns)**	**Latency (#cycles)**	**Fmax (MHz)**
v1_gen	Generic	7.29	86	137.19
v2_SPEC_b1	SPEC b = 1	0	0	NA
v3_SPEC_b3	SPEC b = 3	6.72	13	NA
v4_SPEC_b05	SPEC b = 0.5	5.17	56	NA
v5_SPEC_a1	SPEC a = 1	0	0	NA
v6_SPEC_ab_spf	SPEC a,b SPF	7.01	38	142.59
v7_SPEC_ac_spf_b05	SPEC a,c SPF b = 0.5	6.24	19	NA
v8_MV_b1	MV opt = 1: b = 1	0	0	NA
	MV opt = 2: generic	7.29	80	137.19
v9_MV_b3	MV opt = 1: b = 3	6.72	13	148.83
	MV opt = 2: generic	7.29	81	137.19
v10_MV_b05	MV opt = 1: b = 0.5	5.17	56	193.31
	MV opt = 2: generic	7.29	80	137.19
3***v11_MV_b1_b3**	MV opt = 1: b = 1	0	0	NA
	MV opt = 1: b = 3	6.72	13	148.83
	MV opt = 3: generic	7.29	80	137.19
v12_MV_a1	MV opt = 1: a = 1	0	0	NA
	MV opt = 2: generic	7.23	34	138.23
v13_MV_spf_b05	MV opt = 1: spf b = 0.5	6.24	19	160.33
	MV opt = 2: powf	7.01	34	142.59

Table 5.3: Speedups resulting from comparing the number of latency cycles of each *pow* version against the generic version v1, and the related specialized (SPEC) version.

Version	**Functionality**	**Speedup**	
		v vs v1	**MV vs related SPEC**
v1_gen	Generic	1	-
v2_SPEC_b1	SPEC b = 1	-	-
v3_SPEC_b3	SPEC b = 3	6.62	-
v4_SPEC_b05	SPEC b = 0.5	1.54	-
v5_SPEC_a1	SPEC a = 1	-	-
v6_SPEC_ab_spf	SPEC a,b SPF	2.26	-
v7_SPEC_ac_spf_b05	SPEC a,c SPF b = 0.5	4.53	-
v8_MV_b1	MV opt = 1: b = 1	-	0.00
	MV opt = 2: generic	1.08	1.08
v9_MV_b3	MV opt = 1: b = 3	6.62	1.00
	MV opt = 2: generic	1.06	1.06
v10_MV_b05	MV opt = 1: b = 0.5	1.54	1,00
	MV opt = 2: generic	1.08	1.08
v11_MV_b1_b3	MV opt = 1: b = 1	-	0.00
	MV opt = 1: b = 3	6.62	1.00
	MV opt = 3: generic	1.08	1.08
v12_MV_a1	MV opt = 1: a = 1	-	0.00
	MV opt = 2: generic	2.53	2.53
v13_MV_spf_b05	MV opt = 1: spf b = 0.5	4.53	1.00
	MV opt = 2: powf	2.53	1.12

Compared to the generic version, the number of latency cycles resulting from the specialized versions (*SPEC*) decreased or remained the same for all versions. As anticipated, versions *2* and *5* no longer necessitate any latency cycles for execution, allowing for a significant acceleration. We also highlight the v3_SPEC_b3 version, which shows a speedup of *6.62x* compared to the base version *v1_gen*, indicating that the specialization for the case *b = 3* provides a significant performance boost; the *v6_SPEC_ab_spf* and *v7_SPEC_ac_spf_b05* versions, which show a speedup of *2.26x* and *4.53x*, respectively, compared to the base version *v1_gen*, indicating that the specialization for the use of single-precision floating point numbers (*SPF*) and the replacement of the *pow* function by the *powf* and *sqrtf* functions bring a significant improvement in performance.

As for the number of resources used by the specialized versions, we observe a reduction of over *60 %*, with the preponderance being over 96 %. Given the positive results of the specialized versions compared to the generic version, it can be concluded that the program has the potential to benefit from specialization, and it is therefore appropriate to investigate the *multiversion* optimization strategy.

In comparison to the *v1_gen* version, the *multiversion* (*MV*) versions exhibit an increase or maintenance of specific speedups. As anticipated, option *1* (*opt = 1*) versions *v8_MV_b1*, *v11_MV_b1_b3*, and *v12_MV_a1* do not reach a latency cycle to execute, indicating that the speedup would reach immeasurable values. Additionally, we can highlight the *v12_MV_a1* version with a speedup of *2.53x* and the *v13_MV_spf_b05* variant with a speedup of *1.12x*. Regarding the use of resources, a minor percentage increase was observable when comparing the total resources

Table 5.4: Result number of resources used by each *pow* version after synthesis.

Version	Functionality	BRAM	DSP	FF	LUT	URAM	Total
v1_gen	Generic	30	54	14624	13484	0	28192
v2_SPEC_b1	SPEC b = 1	0	0	0	0	0	0
v3_SPEC_b3	SPEC b = 3	0	11	420	665	0	1096
v4_SPEC_b05	SPEC b = 0.5	0	0	57	249	0	306
v5_SPEC_a1	SPEC a = 1	0	0	0	0	0	0
v6_SPEC_ab_spf	SPEC a,b SPF	7	14	5269	5431	0	10721
v7_SPEC_ac_spf_b05	SPEC a,c SPF b = 0.5	0	0	84	106	0	190
v8_MV_b1	MV b = 1 / generic	30	54	14757	13555	0	28396
v8.1	v1 + v2	30	54	14624	13484	0	28192
v9_MV_b3	MV b = 3 / generic	30	65	15177	14219	0	29491
v9.1	v1 + v3	30	65	15044	14149	0	29288
v10_MV_b05	MV b = 0.5 / generic	30	54	14813	13882	0	28779
v10.1	v1 + v4	30	54	14681	13733	0	28498
v11_MV_b1_b3	MV b = 1 / b = 3 / generic	30	65	15183	14253	0	29531
v11.1	v1 + v2 + v3	30	65	15044	14149	0	29288
v12_MV_a1	MV a = 1 / generic	30	54	14757	13555	0	28396
v12.1	v1 + v5	30	54	14624	13484	0	28192
v13_MV_spf_b05	MV SPF b = 0.5 / generic	7	14	5352	5574	0	10947
v12.1	v1 + v7	30	54	14708	13590	0	28382

used by the generic version to the total resources used by the specific versions that correspond to the specialized versions of each *multiversion*. The exception is *v13_MV_spf_b05*, for which the total number of resources has decreased by *61.17 %* in comparison to the generic version. This is a very satisfactory result given that the total number of resources used in the specific versions that are compatible with this *multiversion* version (*v1* and *v7*) increased by *0.67 %* in comparison to the generic version.

Multiversion is advantageous in this instance because it can incorporate the advantages of the generic and specialized versions while avoiding their disadvantages. The specialized versions are highly optimized for a particular set of conditions, which may render them ineffective under alternate circumstances. The generic version is intended for use across a broad spectrum of conditions, but is less effective in a specific set of conditions than the specialized versions. Using the *multiversion* technique, several specialized versions are built for different sets of conditions and combined into a single version, enabling the program to select the most effective version for each set of conditions. This indicates that the *multiversion* version can compete with the expert version, obtaining virtually the same minimum latency and resource utilisation values as the expert version under conditions that match the optimization parameters.

In this study, versions that use the *powf* or *sqrtf* functions instead of the *pow* and *sqrt* functions, because the function parameters are assumed to be in single-precision float, produced superior results. Due to the difference in precision of calculations conducted, the *powf* function is less resource-intensive than the *pow* function. The *pow* function is designed to operate with double-precision floating-point numbers, which require more bits to represent numbers. This additional precision leads to more complex calculations and necessitates more computational re-

Table 5.5: Comparison of the number of resources used by each *pow* version compared to the generic version (%).

Version	**Functionality**	**Comparison of resources usage with v1 (%)**					
		BRAM	**DSP**	**FF**	**LUT**	**URAM**	**Total**
v2_SPEC_b1	SPEC b = 1	w/o resources	w/o resources	w/o resources	w/o resources	-	w/o resources
v3_SPEC_b3	SPEC b = 3	w/o resources	-79.63	-97.13	-95.07	-	-96.11
v4_SPEC_b05	SPEC b = 0.5	w/o resources	w/o resources	-99.61	-98.15	-	-98.91
v5_SPEC_a1	SPEC a = 1	w/o resources	w/o resources	w/o resources	w/o resources	-	w/o resources
v6_SPEC_ab_spf	SPEC a.b SPF	-76.67	-74.07	-63.97	-59.72	-	-61.97
v7_SPEC_ac_spf_b05	SPEC a.c SPF b = 0.5	w/o resources	w/o resources	-0.99	-0.99	-	-99.33
v8_MV_b1	MV b = 1 / generic	0.00	0.00	0.91	0.53	-	0.72
v8.1	v1 + v2	0.00	0.00	0.00	0.00	-	0.00
v9_MV_b3	MV b = 3 / generic	0.00	20.37	3.78	5.45	-	4.61
v9.1	v1 + v3	0.00	20.37	2.87	4.93	-	3.89
v10_MV_b05	MV b = 0.5 / generic	0.00	0.00	1.29	2.95	-	2.08
v10.1	v1 + v4	0.00	0.00	0.39	1.85	-	1.09
v11_MV_b1_b3	MV b = 1 / b = 3 / generic	0.00	20.37	3.82	5.70	-	4.75
v11.1	v1 + v2 + v3	0.00	20.37	2.87	4.93	-	3.89
v12_MV_a1	MV a = 1 / generic	0.00	0.00	0.91	0.53	-	0.72
v12.1	v1 + v5	0.00	0.00	0.00	0.00	-	0.00
v13_MV_spf_b05	MV SPF b = 0.5 / generic	-76.67	-74.07	-63.40	-58.66	-	-61.17
v12.1	v1 + v7	0.00	0.00	0.57	0.79	-	0.67

sources, including latency cycles and memory usage. In contrast, the *powf* function is optimized for single-precision floating-point numbers, whose numerical representation requires fewer bits. The computations performed by the *powf* function are therefore less complex and demand fewer computational resources than those performed by the *pow* function.

The results of the area-delay product are presented in Table 5.6, reinforcing the conclusions drawn from the previously examined tables.

In summary, the specialized variants of the *pow* function provide significant speedups and a reduction in total resource usage when compared to the generic base version in specific scenarios. In terms of performance and efficiency, the applied specializations are advantageous, as they simplify computations and eliminate superfluous calls to the *pow* function. However, it is essential to keep in mind that these enhancements may differ based on the characteristics of the input data and the optimization options chosen. Performance and calculation precision requirements of the project dictate which variant of the *pow* function should be utilised.

5.2.2 k-Nearest Neighbors

In Chapter 4, the *kNN* benchmark is presented and analysed, together with the relevant versions built from it, taking into account their importance and impact on the study's aims. Tables 5.7 and 5.9 show the results of each of the analysed versions' simulation and synthesis in terms of execution time and resource utilisation. Tables 5.8 and 5.10 allow us to compare these results in terms of speedups and percentage of resources consumed in comparison to the generic version of the algorithm (*v2_gen*). The versions considered are all based on the *WI_K3_F* scenario, presented in Chapter 4, and include the original version (*v1_SPEC_orig*), the generic version (*v2_gen*), the version specialized for *k = 3* (*v3_SPEC_k3*), the version specialized for *k = 3* using the multiversion technique (*v4_MV_k3*), the specialized version for *k = 3*, *N_FEATURES = 43*, *N_TRAINING = 4336*, *N_TESTING = 1082* (*v5_SPEC_kfp*), and the version using the multiversion technique for

Table 5.6: Results of the area-delay product (ADP) of each *pow* version and their comparison (%) with the generic version, v1.

Version	**Functionality**	**Total Resources**	**Latency Cycles**	**ADP**	**Comparison of ADP with v1 (%)**
v1_gen	Generic	28192	86	2424512	**-**
v2_SPEC_b1	SPEC b = 1	0	0	0	-100.00
v3_SPEC_b3	SPEC b = 3	1096	13	14248	-99.41
v4_SPEC_b05	SPEC b = 0.5	306	56	17136	-99.29
v5_SPEC_a1	SPEC a = 1	0	0	0	-100.00
v6_SPEC_ab_spf	SPEC a.b SPF	10721	38	407398	-83.20
v7_SPEC_ac_spf_b05	SPEC a,c SPF b = 0.5	190	19	3610	-99.85
v8_MV_b1	MV opt = 1: b = 1	28396	0	0	-100.00
	MV opt = 2: generic		80	2271680	-6.30
v9_MV_b3	MV opt = 1: b = 3	28192	13	366496	-84.88
	MV opt = 2: generic		81	2283552	-5.81
v10_MV_b05	MV opt = 1: b = 0.5	28779	56	1611624	-33.53
	MV opt = 2: generic		80	2302320	-5.04
v11_MV_b1_b3	MV opt = 1: b = 1	29531	0	0	-100.00
	MV opt = 1: b = 3		13	383903	-84.17
	MV opt = 3: generic		80	2362480	-2.56
v12_MV_a1	MV opt = 1: a = 1	28396	0	0	-100.00
	MV opt = 2: generic		34	965464	-60.18
v13_MV_spf_b05	MV opt = 1: spf b = 0.5	10947	19	207993	-91.42
	MV opt = 2: powf		34	372198	-84.65

the specialized version where *k = 3*, *N_FEATURES = 43*, *N_TRAINING = 4336*, *N_TESTING = 1082* (*v6_MV_kfp*).

Regarding the number of latency cycles resulting from simulations of the specialized versions (*SPEC*), there was a significant decrease for all versions compared to the generic version, with versions *v3_SPEC_k3* and *v5_SPEC_kfp* enhancing the most, as the decrease in latency cycles results in a speedup of approximately *1110x* for both. In terms of the number of resources used by the specialized versions, there is an increase of approximately *343%* between the generic version, *v2_gen*, and the original entirely specialized version, *v1_orig*. Again, versions *v3_SPEC_k3* and *v5_SPEC_kfp* stand out, resulting in an approximately *80%* and *81%* reduction in the total number of resources utilised, which can have a direct impact on lowering expenses and energy efficiency. The results obtained for both versions, both for speedups and percentage of resource utilisation relative to the generic version *v2_gen*, indicate that the transformation with the greatest impact on the algorithm is *k = 3*, given that the only difference between the two versions is the specialization of the variables *N_FEATURES*, *N_TRAINING*, and *N_TESTING*, as verified in Section 4.1.2 of Chapter 4. The results from versions *v3_SPEC_k3* and *v5_SPEC_kfp* are consistent with expectations, considering that in the generic implementation, the algorithm logic must be able to handle any value of *k*, which can lead to computational overload and increased resource usage, whereas in the specialized implementation, the logic is simplified and accelerated for a specific value of *k*, resulting in lower latency and more efficient resource usage. One of the most notable changes made in the specialized versions of *kNN* compared to the generic version was the removal of the histogram in the *kNN_VoteBetweenBest* function, which proved to be a bottleneck in the hardware

Table 5.7: Results after kNN versions simulations of the values referring to the execution time with a target of 5 ns and uncertainty of 1.35 ns. (maximum frequency (Fmax); multiversion (MV); specialized (SPEC))

Version	**Functionality**	**Estimated (ns)**	**Latency (#cycles)**	**Fmax (MHz)**
v1_SPEC_orig	SPEC: original	4.23	95708	236.29
v2_gen	generic	3.64	772545	274.42
v3_SPEC_k3	SPEC: k = 3	3.64	696	274.43
v4_MV_k3	MV opt = 1: k = 3	3.64	696	274.42
	MV opt = 2: Generic	3.64	772551	274.43
v5_SPEC_kfp	SPEC: k && N_FEATURES && N_TRAINING && N_TESTING	3.64	696	274.43
v6_MV_kfp	MV opt = 1: k && N_FEATURES && N_TRAINING && N_TESTING	3.64	696	274.42
	MV opt = 2: generic	3.64	772545	274.43

Table 5.8: Speedups resulting from comparing the number of latency cycles of each kNN version against the kNN generic version v2, and the related specialized (SPEC) version.

Version	**Functionality**	**Speedup**	
		v vs v2	**MV vs related SPEC**
v1_SPEC_orig	SPEC: original	8.07	-
v2_gen	generic	1.00	-
v3_SPEC_k3	SPEC: k = 3	1109.98	-
v4_MV_k3	MV opt = 1: k = 3	1109.98	1.00
	MV opt = 2: generic	1.00	1.00
v5_SPEC_kfp	SPEC: k && N_FEATURES && N_TRAINING && N_TESTING	1109.98	-
v6_MV_kfp	MV opt = 1: k && N_FEATURES && N_TRAINING && N_TESTING	1109.98	1.00
	MV opt = 2: generic	1.00	1.00

Table 5.9: Result number of resources used by each kNN version after HLS synthesis.

Version	**Functionality**	**DSP**	**FF**	**LUT**	**Total**
v1_SPEC_orig	SPEC: original	26	13411	8112	21549
v2_gen	generic	13	2123	2724	4860
v3_SPEC_k3	SPEC: k = 3	2	392	573	967
v4_MV_k3	MV: k = 3 || generic	13	2391	2799	5203
v4.1	v2 + v3	15	2515	3297	5827
v5_SPEC_kfp	SPEC: k && N_FEATURES && N_TESTING && N_TRAINING	2	367	538	907
v6_MV_kfp	MV: (k && N_FEATURES && N_TESTING && N_TRAINING) || generic	13	2224	3065	5302
v6.1	v2 + v5	15	2490	3262	5767

implementation of the algorithm.

Given the considerable differences between the generic version (*v2_gen*) and the specialized versions *v3_SPEC_k3* and *v5_SPEC_kfp*, it is possible to conclude that the specializations benefited the algorithm's implementation. Given this, it was justifiable to study the multiversion optimization technique for both versions in order to optimize and make the hardware implementation of *kNN* flexible.

All of the multiversion (*MV*) versions increase or maintain specific speedups over the generic *v2_gen* version and related specialized versions. Options *1* (*opt* = *1*) of versions *v4_MV_k3* and *v6_MV_kfp*, as expected, stand out by achieving a speedup of roughly *1110x* relative to the generic version, as do the equivalent specialized variants. Still on the *MV* versions, we observe a *7%* increase in total resource utilisation compared to the generic version, *v2_gen*, for the *v4_MV_k3* version and a *9%* increase for the *v6_MV_kfp* version. Despite this slight increase in the percentage of total resource usage, Table 5.10 shows that the results for the sum of the specialized and generic versions that correspond to the specialized versions of each multiversion have a percentage increase that is more than twice as high as the versions with multiversion. This means that versions with multiversion consume fewer resources than the sum of the separate related versions. This is due to resource optimization, resource sharing, and overlap elimination. Some parts of the code, for example, are executed in parallel, or specific processing units are shared. When compared to the some of the separated versions, these strategies allow for more efficient use of available resources, resulting in higher performance and resource savings. As a result, there is an advantage to using multiversion versions, which can compete with the corresponding specialized versions while achieving nearly the same latency and resource utilisation value as the generic version.

The results obtained support the notion that multiversion is preferable in this scenario since it can combine the benefits of generic and specialized versions while avoiding their drawbacks. The specialized versions are highly tailored for a specific set of circumstances but may be unsuccessful in other situations. The generic version is intended for usage in a variety of conditions, but it is less successful in a subset of conditions than the specialized versions. Using the multiversion technique, numerous specialized versions for distinct sets of conditions can be created and combined into a single version, allowing the program to choose the most effective version for each set of

Table 5.10: Comparison of the number of resources used by each kNN version compared to the generic version (%).

Version	**Functionality**	**Comparison of resources usage with v2 (%)**			
		DSP	**FF**	**LUT**	**Total**
v1_SPEC_orig	SPEC: original	100.00	531.70	197.80	343.40
v3_SPEC_k3	SPEC: k = 3	-84.62	-81.54	-78.96	-80.10
v4_MV_k3	MV: k = 3 || generic	0.00	12.62	2.75	7.06
v4.1	v2 + v3	15.38	18.46	21.04	19.90
v5_SPEC_kfp	SPEC: k && N_FEATURES && N_TESTING && N_TRAINING	-84.62	-82.71	-80.25	-81.34
v6_MV_kfp	MV: (k && N_FEATURES && N_TESTING && N_TRAINING) || generic	0.00	4.76	12.52	9.09
v6.1	v2 + v5	15.38	17.29	19.75	18.66

Table 5.11: Results of the area-delay product (ADP) of each kNN version and their comparison (%) with the generic version, v2.

Version	**Functionality**	**Total Resources**	**Latency cycles**	**ADP**	**Comparison of ADP with v2 (%)**
v1_SPEC_orig	SPEC: original	21549	95708	2062411692	-45.07
v2_gen	generic	4860	772545	3754568700	-
v3_SPEC_k3	SPEC: k = 3	967	696	673032	-99.98
v4_MV_k3	MV opt = 1: k = 3	5203	696	3621288	-99.90
	MV opt = 2: generic		772551	4019582853	7.06
v5_SPEC_kfp	SPEC: k && N_FEATURES && N_TESTING && N_TRAINING	907	696	631272	-99.98
v6_MV_kfp	MV opt = 1: (k && N_FEATURES && N_TESTING && N_TRAINING)	5302	696	3690192	-99.90
	MV opt = 2: generic		772545	4096033590	9.09

conditions.

Table 5.11 also includes the area-delay product data, which supports the conclusions drawn from the previous tables. The maximum operational frequency (*Fmax MHz*) stayed close to *274 MHz* in all iterations.

In summary, the results of the HLS synthesis and comparison of various versions of the *kNN* algorithm show that, with the exception of the original *v1_SPEC_orig* version, the specialized versions show performance improvements, providing significant speedups and a decrease in total resource utilisation compared to the generic base version. This can improve energy efficiency and reduce the resulting circuit area. The multiversion optimization technique was successful because it was able to combine the benefits of generic and specialized versions while avoiding their drawbacks. It is crucial to note, however, that these improvements may vary depending on the parameters of the chosen case, the input data, and the optimization options chosen.

5.2.3 Disparity

In Chapter 4, the Disparity Benchmark is presented and analysed, as are the relevant versions developed from it, taking into account their importance and impact on the objectives of the study. Tables 5.12 and 5.14 present the simulation and *HLS* synthesis results of each of the analysed versions in terms of execution time and resource usage. Tables 5.13 and 5.15 allow us to compare these results in terms of speedups and percentage of resources consumed in relation to the generic version of the algorithm. The versions considered are all based on the *SIM* scenario presented in Chapter 4 and include the original/generic version (*v1_orig*), the version specialized for *win_sz = 4* (*v2_SPEC_w4*), the respective version using the multiversion technique (*v3_MV_w4*), the versions that take advantage of the results obtained through the variable monitoring tool (*v4_p4*, *v5_fSAD*, *v6_cSAD*, *v7_II2D* and *v8_all*), the version specialized for the number of rows and columns (*v9_SPEC_r_c*), and the respective version that uses the multiversion technique (*v10_MV_r_c*), and finally the version that specializes *win_sz*, the number of rows and columns (*v11_SPEC_r_c_w*), and the respective multiversion version (*v12_MV_r_c_w*).

In relation to the number of latency cycles that resulted from the simulations of the specialized versions (*SPEC*), it can be seen that all of them present a decrease in comparison to the generic version, *v1_orig*, but the difference is, for the most part, not very significantly different. The most notable versions were *v9_SPEC_r_c* and *v11_SPEC_r_c_w*, where a decrease in latency cycles results in a speedup of approximately *2.4x* for both.

With the exception of *v9_SPEC_r_c* and *v11_SPEC_r_c_w*, which use approximately *26%* and *41%* fewer resources than the generic version, *v1_orig*, there is a slight rise in the number of resources used by the specialized versions in comparison to the generic version, *v1_orig*. *v9_SPEC_r_c*, which specializes in the size of images (*rows x cols*), has a significantly greater impact on the number of latency cycles and resource usage than *v2_SPEC_w4*, which specializes only in the window size, when applied to the Disparity algorithm. By fixing the dimensions of the images, unlike in the *win_sz* specialization, a greater number of loops are optimized, reducing a greater number of operations and enhancing the algorithm's efficiency. This

Table 5.12: Results after Disparity versions simulations of the values referring to the execution time with a target of 10 ns and uncertainty of 2.70 ns. (maximum frequency (Fmax); multiversion (MV); specialized (SPEC))

Version	**Functionality**	**Estimated (ns)**	**Latency (#cycles)**	**Fmax (MHz)**
v1_orig	generic	7.26	937456	137.82
v2_SPEC_w4	SPEC: win_sz = 4	7.26	898038	137.82
v3_MV_w4	MV opt 1: win_sz = 4	7.26	906359	137.82
	MV opt 2: generic	7.26	906359	137.82
v4_p4	Zeros: padarray4	7.26	906360	137.82
v5_fSAD	Zeros: finalSAD	7.26	906360	137.83
v6_cSAD	Zeros: computeSAD	7.26	906360	137.83
v7_II2D	Zeros: IntegralImage	7.26	901768	137.83
v8_all	Zeros: all functions	7.26	901768	137.84
v9_SPEC_r_c	SPEC: row & col	7.26	398280	137.82
v10_MV_r_c	MV opt 1: row & col	7.26	398280	137.82
	MV opt 2: generic	7.26	906360	137.82
v11_SPEC_r_c_w	SPEC: row & col & win_sz	7.26	389925	137.83
v12_MV_r_c_w	MV opt 1: row & col & win_sz	7.26	389941	137.82
	MV opt 2: generic	7.26	906360	137.82

results in a greater direct decrease in latency and an improved utilisation of hardware resources. Therefore, it makes perfect sense that the specialized version that combines both specializations, *v11_SPEC_r_c_w*, performed slightly better than *v9_SPEC_r_c*, which only specializes in image size. Given the significant differences between the generic version, *v1_orig*, and the specialized versions, *v9_SPEC_r_c* and *v11_SPEC_r_c_w*, it is possible to conclude that the algorithm's implementation benefites from the specializations. In light of this, it was appropriate to investigate the multiversion optimization technique for both versions in order to optimize and increase the flexibility of the hardware implementation.

All multiversion versions (*MV*) reduce the number of latency cycles necessary for their operation in comparison to the original version, *v1_orig*. As anticipated, the options *1* (*opt* = *1*) of the *v10_MV_r_c* and *v12_MV_kfp* versions achieve a speedup of approximately *2.4x* compared to the generic version, just like the equivalent specialized variants. In addition, the *MV* versions exhibit a rise in total resource usage compared to the original version, *v1_orig*. Despite this increase in the percentage of total resource usage, Table 5.15 shows that the results for the sum of the specialized and generic versions corresponding to the specialized versions of each multiversion have a greater percentage increase in resource usage relative to the generic version than versions with multiversion. This indicates that the multiversion versions utilise fewer resources than the sum of the resources utilised by the individual versions. This is due to the optimization of resources, the sharing of resources, and the elimination of overlaps. Some parts of the code, for instance, are executed in parallel, and certain processing units are shared. These strategies make more efficient use of available resources than some of the distinct versions, resulting in improved performance and cost savings. Consequently, there is an advantage to utilising multiversion versions, which can compete with their corresponding specialized counterparts and achieve nearly the same la-

Table 5.13: Speedups resulting from comparing the number of latency cycles of each Disparity version against the Disparity generic version v1, and the related specialized (SPEC) version.

Version	**Functionality**	**Speedup**	
		v vs v1	**MV vs related SPEC**
v1_orig	generic	1.00	-
v2_SPEC_w4	SPEC: win_sz = 4	1.04	-
v3_MV_w4	MV opt 1: win_sz = 4	1.03	0.99
	MV opt 2: generic	1.03	1.03
v4_p4	Zeros: padarray4	1.03	-
v5_fSAD	Zeros: finalSAD	1.03	-
v6_cSAD	Zeros: computeSAD	1.03	-
v7_II2D	Zeros: IntegralImage	1.04	-
v8_all	Zeros: all functions	1.04	-
v9_SPEC_r_c	SPEC: row & col	2.35	-
v10_MV_r_c	MV opt 1: row & col	2.35	1.00
	MV opt 2: generic	1.03	1.03
v11_SPEC_r_c_w	SPEC: row & col & win_sz	2.40	-
v12_MV_r_c_w	MV opt 1: row & col & win_sz	2.40	1.00
	MV opt 2: generic	1.03	1.03

tency and resource utilisation as the generic version. As mentioned in Section 5.2.2, the obtained results support the notion that multiversion is preferable in this scenario because it can combine the benefits of the generic and specialized versions while avoiding their drawbacks.

There were no small improvements in latency cycles in the versions that deal with the treatment of zeros, *v4_p4* through *v8_all*, as compared to the generic version, *v1_orig*. There was just a slight rise in speedups, indicating a minor boost in performance. Although the multiversion technique used in these versions requires the use of conditionals throughout the runtime to select the appropriate behaviour based on parameter values, adding complexity to the code, it could have resulted in an increase in latency cycles due to the conditional checks required during execution, which did not occur. Furthermore, in terms of total resource usage, there was a slight increase in the number of resources compared to the generic version, due to the need to allocate additional memory to store information related to the specific treatment of zeros, but it was significantly lower compared to the other *MV* versions, which involve more significant code modifications and more intensive optimizations. These results are satisfactory given that these versions must use the multiversion approach to function correctly since the parameter values are assigned at runtime.

These findings emphasise the need of carefully determining which optimizations are most effective in a specific situation. For this benchmark and scenario, although the benefit was not as expressive as in the *MV* versions, and there was also a slight increase in the number of resources, it was possible to reduce the number of latency cycles, indicating an improvement in the algorithm's efficiency.

Table 5.16 also provides the area-delay product data, which validates the preceding tables' results. In all iterations, the maximum operating frequency (*Fmax MHz*) remained close to *138 MHz*.

In summary, the results of the synthesis and comparison of various versions of the disparity

Table 5.14: Result number of resources used by each Disparity version after synthesis.

Version	**Functionality**	**BRAM**	**DSP**	**FF**	**LUT**	**Total**
v1_orig	generic	255	53	7839	10289	18436
v2_SPEC_w4	SPEC: win_sz = 4	288	62	8268	10308	18926
v3_MV_w4	MV: winsz ‖ generic	319	65	10196	13136	23716
v3.1	v1 + v2	543	115	16107	20597	37362
v4_p4	Zeros: padarray4	319	55	8047	10521	18942
v5_fSAD	Zeros: finalSAD	319	55	8371	10807	19552
v6_cSAD	Zeros: computeSAD	319	55	8111	10595	19080
v7_II2D	Zeros: IntegralImage	319	58	8379	11140	19896
v8_all	Zeros: all functions	319	58	8767	11524	20668
v9_SPEC_r_c	SPEC: row & col	302	36	5105	8211	13654
v10_MV_r_c	MV: row ‖ col	319	78	12301	17764	30462
v10.1	v1 + v9	557	89	12944	18500	32090
v11_SPEC_r_c_w	SPEC: row & col & win_sz	288	22	3738	6888	10936
v12_MV_r_c_w	MV: row & col & winsz ‖ generic	319	69	11397	16741	28526
v12.1	v1 + v11	543	75	11577	17177	29372

Table 5.15: Comparison of the number of resources used by each Disparity version compared to the generic version (%).

Version	**Functionality**	**Comparison of resources usage with v1 (%)**				
		BRAM	**DSP**	**FF**	**LUT**	**Total**
v2_SPEC_w4	SPEC: win_sz = 4	12.94	16.98	5.47	0.18	2.66
v3_MV_w4	MV: winsz ‖ generic	25.10	22.64	30.07	27.67	28.64
v3.1	v1 + v2	112.94	116.98	105.47	100.18	102.66
v4_p4	Zeros: padarray4	25.10	3.77	2.65	2.25	2.74
v5_fSAD	Zeros: finalSAD	25.10	3.77	6.79	5.03	6.05
v6_cSAD	Zeros: computeSAD	25.10	3.77	3.47	2.97	3.49
v7_II2D	Zeros: IntegralImage	25.10	9.43	6.89	8.27	7.92
v8_all	Zeros: all functions	25.10	9.43	11.84	12.00	12.11
v9_SPEC_r_c	SPEC: row & col	18.43	-32.08	-34.88	-20.20	-25.94
v10_MV_r_c	MV: row ‖ col	25.10	47.17	56.92	72.65	65.23
v10.1	v1 + v9	118.43	67.92	65.12	79.80	74.06
v11_SPEC_r_c_w	SPEC: row & col & win_sz	12.94	-58.49	-52.32	-33.05	-40.68
v12_MV_r_c_w	MV: row & col & winsz ‖ generic	25.10	30.19	45.39	62.71	54.73
v12.1	v1 + v11	112.94	41.51	47.68	66.95	59.32

Table 5.16: Results of the area-delay product of each Disparity version and their comparison (the generic version, v1.

Version	Functionality	Total Resources	Latency cycles	ADP	Comparison of ADP with v1 (%)
v1_orig	generic	18436	937456	17282938816	0.00
v2_SPEC_w4	SPEC: win_sz = 4	18926	898038	16996267188	-1.66
v3_MV_w4	MV opt 1: win_sz = 4	23716	906359	21495210044	24.37
	MV opt 2: generic		906359	21495210044	24.37
v4_p4	Zeros: padarray4	18942	906360	17168271120	-0.66
v5_fSAD	Zeros: finalSAD	19552	906360	17721150720	2.54
v6_cSAD	Zeros: computeSAD	19080	906360	17293348800	0.06
v7_II2D	Zeros: IntegralImage	19896	901768	17941576128	3.81
v8_all	Zeros: all functions	20668	901768	18637741024	7.84
v9_SPEC_r_c	SPEC: row & col	13654	398280	5438115120	-68.53
v10_MV_r_c	MV opt 1: row & col	30462	398280	12132405360	-29.80
	MV opt 2: generic		906360	27609538320	59.75
v11_SPEC_r_c_w	SPEC: row & col & win_sz	10936	389925	4264219800	-75.33
v12_MV_r_c_w	MV opt 1: row & col & win_sz	28526	389941	11123456966	-35.64
	MV opt 2: generic		906360	25854825360	49.60

algorithm show that, with the exception of version *v2_SPEC_w4*, the specialized versions show performance improvements, providing significant increases in speedup and a decrease in total resource usage relative to the generic base version. This can enhance energy efficiency, lower the resultant circuit size, and cut overall expense. The multiversion optimization approach was effective because it was able to mix the benefits of the generic and specialized versions while avoiding their limitations. It is important to note, however, that these gains may vary based on the case specifications, input data, and optimization strategies used.

5.3 Analysis of the Relationship between Degree of Similarity and Performance in Multiversion Technique

During the development of this study, the opportunity arose to research whether there is a correlation between the degree of similarity between different specialized and generic versions and the number of resources required for a version using the multiversion technique for the same versions. This section aims to present and analyse the results obtained from this research.

5.3.1 Relative Percentage of Resources Used

As the goal of this study is to explore if there is a correlation between the degree of code similarity between different specialized and generic versions and the amount of resources needed to make a multiversion version of them, it wouldn't make sense to do the same calculation for all multiversion versions against the same generic version, as was done in the previous section. This is because there are multiversion versions made up of only very specialized versions, whose combined resources use much less than those of a generic version. We wouldn't have an adequate way

Table 5.17: Results obtained for *Disparity* from the analysis of the degree of similarity between each of the two possible combined individual versions and the corresponding results of the multiversion versions

Version	**Functionality**	**Similarity**	**Total Resources SPEC 1**	**Total Resources SPEC 2**	**Total Resources MV**	**Total Resources SUM (SPECs)**	**MV Relative % Resources (min SPEC)**	**Rank Similarity**	**Rank Total Resources MV**	**Rank MV Relative % Resources (min SPEC)**
v13_MV_9_11	v9 + v11	0.15	13654	10936	19109	24590	33.24	6	6	5
v10_MV_1_9	v1 + v9	0.22	18436	13654	30462	32090	52.38	5	1	3
v3_MV_1_2	v1 + v2	0.25	18436	18926	23716	37362	14.13	4	5	6
v12_MV_1_11	v1 + v11	0.25	18436	10936	28526	29372	59.89	3	3	1
v17_MV_2_11	v2 + v11	0.26	18926	10936	26824	29862	53.20	2	4	2
v16_MV_2_9	v2 + v9	0.30	18926	13654	28866	32580	46.69	1	2	4

to compare a multiversion version with a generic version since the specialized versions that constitute the multiversion version are selected to improve performance by using their own features.

For this study, a metric was used to figure out the relative percentage of resources used by the multiversion version compared to the smaller number of resources used by the corresponding individual versions over the total number of resources used by all individual versions. This metric assesses the relative gain in terms of resource reductions that the multiversion version offers in comparison to the individual specialized versions. The higher the metric's final value, the greater the resource savings, and hence the better the performance of the multiversion version in terms of efficient resource usage.

5.3.2 Presentation and Analysis of the Results

The results obtained for the *Disparity* and *kNN* benchmarks, respectively, from the analysis of the degree of similarity between each of the two possible combined individual versions and the corresponding results of the multiversion versions are presented in Tables 5.17 and 5.19. Tables 5.18 and 5.20 present the results obtained for the *Disparity* and *kNN* benchmarks, respectively, of the multiversion versions suggested by the clusters created by the *AC* tool through the existing individual versions.

The *"Similarity"* column indicates the degree of similarity between the two individual versions that led to the multiversion version. The closer the number is to *1*, the less similar it is. The *"MV Relative % Resources"* column displays the relative percentage of resources used by the multiversion version relative to the shortest number of resources used by the corresponding combined versions as a proportion of the total number of resources used by all combined versions.

Table 5.18: Results obtained for *Disparity* from clusters resulting from the analysis of the degree of similarity between each of the two possible combined individual versions and the corresponding results of the multiversion versions

Version	**Functionality**	**Similarity**	**Total Resources SPEC 1**	**Total Resources SPEC 2**	**Total Resources SPEC 3**	**Total Resources SPEC 4**	**Total Resources MV**	**Total Resources SUM (SPECs)**	**MV Relative % Resources (min SPEC)**	**Rank Similarity**	**Rank Total Resources MV**	**Rank MV Relative % Resources (min SPEC)**
v13_MV_9_11	v9 + v11	0.15	13654	10936	-	-	19109	24590	33.24	3	3	3
v14_MV_9_11_1	v9 + v11 + v1	0.25	13654	10936	18436	-	35536	43026	57.17	2	2	1
v15_MV_9_11_1_2	v9 + v11 + v1 + v2	0.30	13654	10936	18436	18926	45416	61952	55.66	1	1	2

Table 5.19: Results obtained for *kNN* from the analysis of the degree of similarity between each of the two possible combined individual versions and the corresponding results of the multiversion versions

Version	Functionality	Similarity	Total Resources SPEC 1	Total Resources SPEC 2	Total Resources MV	Total Resources SPEC 1 + SPEC 2	MV Relative % Resources (min SPEC)	Rank Similarity	Rank Total Resources MV	Rank MV Relative % Resources (min SPEC)
v8_MV_2_7	v2 + v7	0.11	4860	4670	9973	9530	55.65	10	1	5
v13_MV_3_5	v3 + v5	0.13	967	907	1869	1874	51.33	9	10	6
v9_MV_1_2	v1 + v2	0.24	21549	4860	6181	26409	5.00	8	2	7
v12_MV_1_7	v1 + v7	0.25	21549	4670	5906	26219	4.71	7	3	8
v6_MV_2_5	v2 + v5	0.32	4860	907	5302	5767	76.21	6	5	2
v4_MV_2_3	v2 + v3	0.36	4860	967	5203	5827	72.70	5	6	4
v11_MV_1_5	v1 + v5	0.34	21549	907	1956	22456	4.67	4	9	9
v14_MV_3_7	v3 + v7	0.34	967	4670	5447	5637	79.47	3	4	1
v15_MV_5_7	v5 + v7	0.35	907	4670	4962	5577	72.71	2	7	3
v10_MV_1_3	v1 + v3	0.37	21549	967	2016	22516	4.66	1	8	10

From Tables 5.17 and 5.19, we can verify that, in some instances, there might be a correlation between the similarity rank and the rank of the multiversion versions' percentage resource usage. When the degree of similarity between individual versions increases (higher similarity rank), the total number of resources in the corresponding multiversion versions decrease (higher rank). Similarly, in some instances, when the degree of similarity between the individual versions increases (higher similarity rank), the relative percentage of resources used by the multiversion versions increases, resulting in greater resource savings.

This pattern is also evident in the data presented in Tables 5.18 and 5.20. This could indicate that the use of the clusters suggested by the *AC* tool would be a good indicator of what multiversion versions would be more advantageous to create. However, the conducted study does not provide enough case studies to conclude that such a correlation exists. In the same table, we can also see that versions that combine more than two versions typically achieve a relative percentage of saved resources that is comparable to or higher than versions that only combine two versions, such as version *v15_MV_9_11_1_2* in table 5.18, which combines versions *v9, v11, v1,* and *v2* and has a relative percentage of saved resources of *55.66 %*. This suggests that even when combining more than two versions with a certain degree of similarity, we can still have a significant effect on the resource utilisation of the multiversion versions.

In addition, we can confirm that the results of all tables indicate that the use of resources was significantly reduced in all multiversion versions generated compared to the respective combined versions, particularly for the versions generated from the clusters suggested by the *AC* tool. Not

Table 5.20: Results obtained for *kNN* from clusters resulting from the analysis of the degree of similarity between each of the two possible combined individual versions and the corresponding results of the multiversion versions

Version	Functionality	Similarity	Total Resources SPEC 1	Total Resources SPEC 2	Total Resources SPEC 3	Total Resources SPEC 4	Total Resources SPEC 5	Total Resources MV	Total Resources SUM (SPECs)	MV Relative % Resources (min SPEC)	Rank Similarity	Rank Total Resources MV	Rank MV Relative % Resources (min SPEC)
v8_MV_2_7	v2 + v7	0.11	4860	4670	-	-	-	9973	9530	55.65	4	2	1
v13_MV_3_5	v3 + v5	0.13	967	907	-	-	-	1869	1874	51.33	3	4	2
v16_MV_1_2_7	v1 + v2 + v7	0.25	21549	4860	4670	-	-	9129	31079	14.35	2	3	4
v17_MV_2_7_3_5_1	v2 + v7 + v3 + v5 + v1	0.35	4860	4670	967	907	21549	12376	32953	34.80	1	1	3

only does this imply that, depending on the situation, it may be advantageous to employ techniques such as multiversion, but also that similarity detection tools may be useful in determining which versions to combine.

Ideally, the multiversion technique would be able to identify the presence of similar patterns and structures and eliminate redundancies, thereby making it possible, among others, to reduce the required amount of resources when the combined versions are extremely similar and contain numerous identical code portions. The results obtained thus far indicate that the degree of similarity between the combined versions could potentially have an effect on the use of resources in versions with multiversion, but they are insufficient to demonstrate a causal relationship between the two. This lack of correlation can be attributed to the nature of the tool employed, which measured the degree of similarity based on the entire code of the combined versions rather than the portions of the code that may actually result in the consumption of more resources. When optimizing code for *FPGAs*, it is essential to consider not only the superficial similarity of the code, but also its intrinsic properties, which may impact resource utilisation. These characteristics include memory access patterns, arithmetic operations, and the use of multiplexers, among other factors that have a direct effect on *FPGA* resource utilisation.

In consideration of these limitations, it is crucial to interpret the results obtained thus far with caution and to continue investigating more exhaustive approaches for evaluating the relationship between the similarity of specialized versions and resource utilisation by multiversion versions. To gain a more comprehensive understanding of the relationship between the degree of similarity and resource usage by multiversion versions, it is necessary to develop a strategy that incorporates more granular metrics. This will allow for a more comprehensive understanding of the factors that influence the performance and efficiency of multiversion versions.

5.4 Value Counter Monitor: Extended Version

We extended our colleague Pedro Pinto's "*monitors*" library, which he had built in the *SPECS* group, as mentioned in Section 3.5 of Chapter 3. This section's major purpose is to provide and analyse the outcomes of the library's extended version.

The *SIM* scenario and the disparity algorithm's "*diff*" variable were used to test the implementations. This variable was chosen because of its large range of values and number of assignments, making it perfect for studying and comparing the various replacement policies. There are *35264* assignments to the variable "*diff*" in total. The tests were carried out with a maximum of *10* hashtable entries.

Figure 5.1 and Table 5.21 show the results of simulating the disparity algorithm for each substitution policy without limiting the value range, as well as the resulting parameter values indicating the total number of values assigned to the monitored variable that entered the hashtable and the total number of values substituted in the table. The "ORIG" column contains the original library "monitors" values prior to the changes.

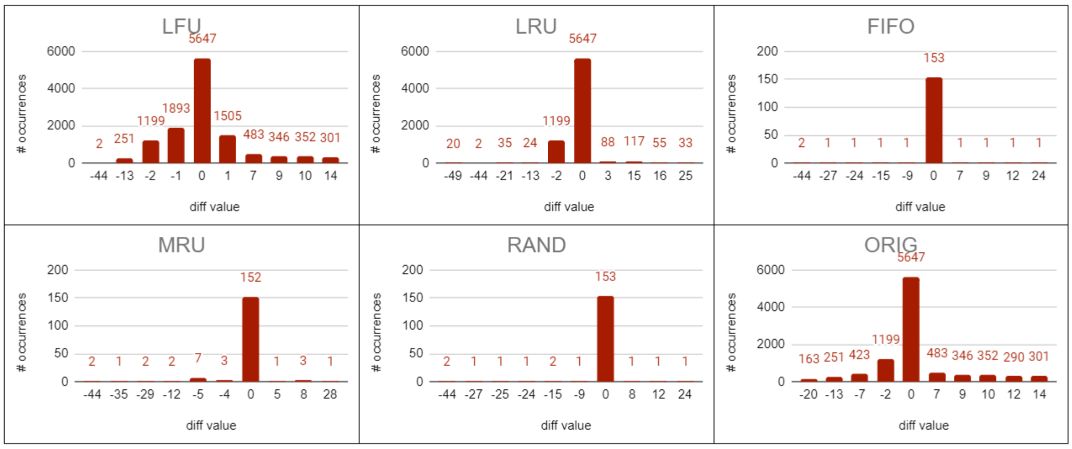


Figure 5.1: Results of executing the disparity algorithm for each substitution policy without limiting the value range

Using these tables, we can see that each replacement policy has a distinct behaviour, resulting in different value distribution patterns. The *FIFO* approach results in a fairly uniform distribution of occurrences with no apparent pattern in the "*diff*" values. The *LFU* policy has a higher concentration of occurrences in a smaller number of distinct values, indicating that these values are frequently read and stored in the hashtable. For values recently allocated to the "*diff*" variable, the *MRU* policy reflects a higher frequency of occurrences. The *RAND* (Random) policy produces a random distribution of occurrences with no discernible pattern in the "*diff*" values. The value *0* is the value having the greatest number of occurrences in all policies. The substitution differences demonstrate the efficiency of each substitution policy in maintaining the most relevant values in the hashtable. All of the substitution rules result in fewer overall substitutions than the original library. The *LRU*, *LFU*, and *MRU* policies are intended to keep more relevant data in the hashtable, which explains why they have the highest value, minimising the total number of replacements and implying a higher efficiency in hashtable use. The *FIFO* policy replaces the oldest values first, regardless of their frequency of usage, which explains why there are more replacements.

Figure 5.2 and Table5.22 provide the outcomes of the same procedure but with a range of values supplied to the monitored variable ranging from *-10* to *10*. These numbers were chosen based on the study of the variable "*diff*" performed in Section 4.2.1, where the original library "monitors" was used with a hashtable containing a maximum of 100 entries. This analysis produced a histogram that revealed a larger concentration of occurrences among these values. We direct our

Table 5.21: Resulting parameter values indicating the total number of values assigned to the monitored variable that entered the hashtable and the total number of values substituted in the table of simulating the disparity algorithm for each substitution policy

Parameters	**FIFO**	**LRU**	**LFU**	**MRU**	**RAND**	**ORIG**
# Total Values	24425	24063	22395	23785	24496	25819
# Total Substitutions	24415	24053	22385	23775	24486	25809

Table 5.22: Resulting parameter values indicating the total number of values assigned to the monitored variable that entered the hashtable and the total number of values substituted in the table of simulating the disparity algorithm for each substitution policy with a value range between [-10,10]

Parameter	FIFO	LRU	LFU	MRU	RAND	ORIG
# Total Values	6458	6735	5344	6428	6468	7564
# Total Substitutions	6448	6725	5334	6418	6458	7554

analysis to a certain range of interest by reducing the range of values provided to the monitored variable, where the original library revealed a larger concentration of occurrences. This allows us to investigate the performance of the substitution strategies more precisely and compare the results across library versions.

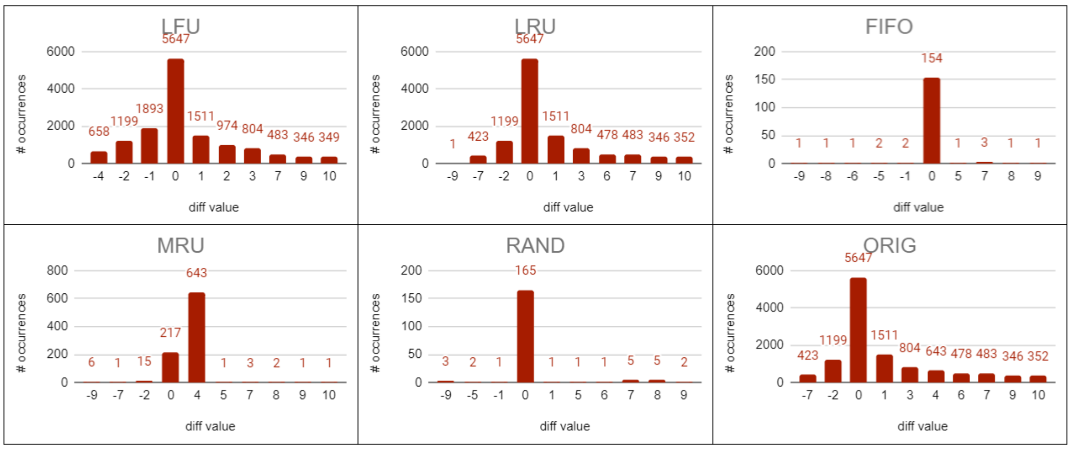


Figure 5.2: Results of simulating the disparity algorithm for each substitution policy with a value range between [-10,10]

We can see from these tables that some policies, like *LFU* and *MRU*, have a higher number of occurrences for values close to zero, but others, like *FIFO* and *LRU*, have a more uniform distribution over the range. As shown in Table 5.21, the quantities of assigned values and substitutions are similar across substitution policies. This shows that, even with the limited value range, all policies performed similarly in terms of the number of substitutions.

In comparison to the results obtained without limiting the range of values assigned to the monitored variable, we find that by limiting the range of values assigned to the monitored variable, the results focus on the most relevant and significant values for the analysis, and the number of entries and substitutions in the table decreases. This leads to a more accurate and direct understanding of the behaviour of replacement policies within that specific range, as well as potential savings in computational resources, both storage and processing.

It is crucial to note, however, that whether a limited or infinite range is used depends on the aims of the analysis and the characteristics of the system under study. Each approach provides unique insights and may be appropriate in specific situations. The replacement policy that is utilised in a system is determined by a number of criteria, including system characteristics, performance needs, resource availability, and unique hardware constraints. Because each application is unique, there is no single policy that is best in all circumstances. Experiments and extensive analysis are recommended to evaluate the performance of each policy with respect to the system requirements.

This study of the results in the tables helped us to understand how the various substitution policies behave in terms of value distribution and substitution quantity, both in situations with and without range restrictions. This aids in evaluating the relative performance of substitute policies and comprehending how they can alter the outcomes of work accomplished. The functions introduced in the "monitors" library allowed for the exploration of various data management approaches and the improvement of monitoring efficiency, making it more versatile and adaptive to a variety of changing monitoring requirements and circumstances.

5.5 Analysis on code changes to be made for automation

There are numerous benefits to automating an approach, including efficiency, accuracy, and consistency. In addition, automation permits scalability, cost reduction, and enhanced work quality. By eliminating repetitive manual duties, developers are able to concentrate on higher-value activities, thereby increasing their productivity and supporting innovation.

In order to automate the process of code specialization for *FPGAs*, a comprehensive analysis of the required modifications to the original source code is required. These modifications are required to acquire optimized versions for the *FPGA* platform of interest. In this section, the primary considerations and modifications that must be made to the original code in order to automate the process of specialization are discussed.

5.5.1 Specialized Versions

Identifying critical regions that may be candidates for specialization is the first step in analysing code changes. Typically, these regions contain portions of code that consume a considerable quantity of resources or cause performance bottlenecks. Through static analysis and/or profiling techniques, these regions can be identified and their suitability for specialization determined. In addition to the previously mentioned performance analysis tools, a library capable of monitoring variables can be of great value. This library enables the monitoring of variable behaviour during code execution and the identification of access and usage patterns. This data is helpful for identifying opportunities for specialization, such as eliminating redundant computations or optimizing data structures.

We observed that for both benchmarks, algorithm-critical parameters with constant values were identified, increasing the possibility of automating the specialization procedure. For example, the value of *k* for *kNN*, as presented in Section 4.1.2, and the values of rows and columns for Disparity, as presented in Section 4.2.1. Given that the value of these parameters is always set at the beginning of the algorithm and used later for various operations such as loop iteration, the development of a code transformation capable of traversing the algorithm's source code and modifying the variables according to the desired values for the various existing scenarios would be advantageous. To specialize, in many cases requires to replace all instances of a variable with a particular value.

There are, however, cases that necessitate manual intervention, such as some of the transformations performed for *kNN* algorithm.

5.5.2 Multiversion Versions

The process of selecting the versions that would comprise the multiversion versions was one of the stages of this project. Initially, this decision was based solely on the degree of specialization between specialized and generic versions. Later, the opportunity arose, based on the article [76], to determine if there is a correlation between the degree of similarity between various specialized versions and the number of resources required for a version that uses the multiversion technique for the same versions. The remaining multiversion versions were created with the tool's recommendations for detecting similarity between versions in mind.

If the results of this study indicated that there was a correlation between the degree of similarity between different specialized versions and the number of resources required for a version that uses the multiversion technique for the same versions, it would be beneficial to use the results of the tool for similarity detection between versions, such as the degree of similarity between versions, and attempt to automate this process. The plan is to employ a strategy comparable to that described in [59, 62]. Define a threshold for the degree of similarity between versions, for instance, to determine which versions would result in an efficient multiversion, and use a script or tool capable of reading this data to perform the automated analysis, classify the parameter configurations, and select those that meet the defined criteria. The outcome would be a recommended inventory of parameter settings for the multiversion version.

Depending on the selection criteria and the data involved, the actual process of automation may be more complex. In addition, it is essential to continuously validate and adjust the automated process based on the system's actual results and feedback.

5.6 Global Approach Evaluation

Essential to identifying specific contributions, validating their significance, identifying limitations, and contextualising the work within the scientific field is assessing the overall performance of the developed approach and comparing it to the state-of-the-art of this work. This analysis provides a solid foundation for highlighting the approach's value and originality and communicating its significance to the research community. This section seeks to perform a comprehensive evaluation of our work's structure, identify its limitations, and compare it to the state of the art presented in Chapter 2.

Our work focuses on exposing, exploring, and developing efficient specialization and multiversion techniques for application code in an *HLL*, in particular using *HLS* tools and targeting

FPGAs. Most existing approaches investigate projects aimed at optimizing code for software, such as [59, 62, 64, 66, 67, 28], or on optimizing code using *HLS* tools and targeting *FPGAs* that only use specialization and multiversion techniques as a supplement. To simplify this task, we proposed an approach that not only enables the identification of a series of steps for implementing our solution, but also the evaluation of its efficiency.

5.6.1 Benchmark Selection

Evaluation of the selection of benchmarks is essential to the development of a research project. As with [42], the selected benchmarks belong to the image processing application domain. Although they were sufficient for evaluating the performance of our approach, we discovered a deficiency in the representation of various types of operations, which limited the investigation of prospective specializations. A pertinent example is the identification of variables that can be assigned a zero value or have a higher probability of being zero, which would simplify processes and result in better performance. Contrary to what was observed in [75], our results did not demonstrate the same level of gratification because this specialization could not be applied to a broad range of situations. Therefore, it would be advantageous to explore more benchmarks, as in [42, 36, 6], so that one can explore a broader spectrum of specializations. This strategy could uncover patterns that can be automated, thereby providing valuable insights for future development.

5.6.2 Profiling and Monitoring Variable Values

Similar to [42, 62], we used static runtime and analysis to identify specializations and generate optimized code. The benchmark disparity is more complex and extensive, necessitating the use of profiling tools to facilitate the identification of the critical code regions and the evaluation of each version's runtime consumption. Profiling made it possible to acquire detailed information about the performance of the code during execution and to identify the functions that consumed the most time. With this information, it was possible to conduct specific optimizations at the code's critical points, resulting in an increase in its efficiency. Furthermore, following the same idea used in [36], in which the *Daikon* using *Kvasir* was used for dynamic invariant detection, we used the "monitors" library.

The library's "monitors", Section 3.5, allowed us to analyse the behaviour of variables during the program's execution and to identify patterns or trends that influence the selection of specialization strategies. This library uses a hashtable to store variable values and assignment counts. This strategy of storing pertinent information during program execution is consistent with the concept of partial evaluation described in [50], in which computation and/or variable characteristics are stored for later reuse. Both strategies seek to reduce computational load and enhance program performance.

5.6.3 Specialized and Multiversion Versions

On the foundation of the data obtained from profiling (*Gprof*) and variable monitoring ("monitor" library), statically specialized versions of each benchmark were developed. This approach of identifying specific features of the code and optimizing based on them is comparable to *HeteroRefactor* [36], in which specific dynamic invariants, such as the requisite bit-width for variables, are identified. Our analysis enabled us to determine the most advantageous specializations and assess their effect on performance and resource usage. Most of these transformations are static constant specializations that replace variable references with their corresponding constant values during compilation or code interpretation.

There are, however, certain cases that require manual intervention, such as the transformations conducted for version *v3_SPEC_k3* of the *kNN* algorithm. The modifications made to the *kNN* algorithm's *kNN_UpdateBest_Caching* function, which can be seen in Listings 4.3 and 4.4, simplified the code by eliminating the loop used to determine the *k* nearest neighbours and replacing it with a sequence of specific direct comparisons when $k = 3$. In addition, the function now returns only the distance value of the most distant discovered point, which is the second worst among the *k* nearest neighbours, thereby eradicating the need to store and sort the *k* nearest neighbours. The code for the function *kNN_VoteBetweenBest* was altered by removing the histogram and the loops used to tally the class votes. These modifications can be analysed in Listings 4.5 and 4.6. In spite of this, the function conducts a series of experiments using only the three closest points ($k = 3$) to determine the most popular class. By setting *k* equal to three and removing the histogram and for loops, the code can be simplified, leading to more efficient hardware implementations. This type of simplification and specialization is also utilised in *HeteroRefactor*, which rewrites recursive data structures as arrays of finite size and adjusts floating-point operations with variable width.

Similar to those used in the article [6], metrics such as speedup, percentage of resources used, and ADP were applied to generic variants of each benchmark. These metrics facilitate the evaluation and analysis of results, enabling a quantitative comparison of the performance, resource utilisation, and efficacy of the specialized version and the generic version. The comparison with the generic version provides a benchmark for evaluating the impact of specializations, which is an advantage. The standard implementation of the generic version is not optimized for any particular scenario. By comparing this version to the specialized version, it is possible to assess the performance increase, efficiency, and precision attained by the specialization. After synthesis, the results of these specializations are very positive, especially in cases where the specialized variables have a greater impact, constant presence, or influence on the algorithms, such as the case of *k* for *kNN*, as seen in Section 4.1.2, and the number of rows and columns, for the disparity case, in Section 4.2.2. These specializations eliminated the need to retrieve the variable's value at runtime, which can result in more efficient and quicker code overall. specialization of variables into constants is one strategy used to reduce program runtime and enhance its performance.

The transformations that resulted in the versions presented in Section 4.2.2 do not produce satisfactory synthesis results compared to the others, most likely due to the small number of operations that could be simplified, such as multiplications, which reduce the use of additional resources and computational load overall. It is believed that for a benchmark with more operations that could benefit from that optimizations, the outcomes would be much more significant.

In contrast to [59, 64, 66, 67] approaches, the multiversions whose synthesis results were analysed were generated statically from combinations of the specialized version and the respective generic version of the algorithm and each version is selected at runtime based on the "opt" variable. The results were positive overall as, despite a slight increase in the percentage of total resource utilisation, they demonstrate that the results of the sum of the specialized and generic versions that correspond to the specialized versions of each multiversion have a percentage increase that is more than twice that of the multiversion versions, while achieving nearly the same latency and resource utilisation as the generic version. As a result of resource sharing and the reduction of overlaps, there was resource optimization.The obtained results support the notion that multiversion is preferable in these circumstances because it can combine the benefits of generic and specialized versions while avoiding their drawbacks. It is essential to note, however, that these enhancements may vary depending on the case parameters, input data, and optimization options chosen.

5.6.4 Design Flow

The design flow used was similar to that of the studies carried out in [36, 42, 6], with the exception of the tools used. In order to analyse the impact of the performance that each version would have on the target *FPGA* board, *PYNQ™-Z2* , the *Vitis HLS* tool, version *2022.2*, were used. The tool enabled logic synthesis and the application of additional optimization techniques to enhance the performance, footprint, and energy efficiency of the synthesised circuit. The process of high-level synthesis entails an iterative cycle in which the design is refined by repeating the previous steps.

5.6.5 Relationship between Degree of Similarity and Performance in Multiversion Technique

Using the *AC* tool [77], a correlation between the resources resulting from the *HLS* synthesis of multiversion versions and the degree of similarity between the versions that constitute them was investigated. This research was motivated by [76]. To conduct this study, a metric was employed that computes the relative percentage of resources used by the multiversion version relative to the minimum number of resources used by the specialized versions over the total number of resources used by all specialized versions. This metric enables a comparison of the relative resource savings offered by the multiversion version versus the individual specialized versions.

This study provided some understanding on the effectiveness of multiversion versions and the significance of specialized version selection.

The results obtained thus far, imply that the degree of similarity between the specialized versions may have an effect on resource utilisation in the multiversion versions, but are insufficient to prove a direct correlation between the two.

This lack of correlation, in addition to possibly resulting from the absence of case studies, can be attributed to the nature of the utilised tool, which measured the degree of similarity based on the entire code of the specialized versions and not the portions of the code that may actually result in the consumption of more resources. Although the results are not ideal, it was discovered that versions combining more than two versions can achieve a relative percentage of saved resources nearly equal to or higher than versions combining only two versions, indicating that even when combining more than two specialized versions with up to a certain degree of similarity, we can still have a significant impact on the version's resource usage. In addition, the results indicate that there was a significant reduction in resource usage between the multiversion versions and the respective specialized versions, particularly for the versions derived from the clusters suggested by the *AC* tool. Not only does this imply that, depending on the circumstance, it may be advantageous to employ techniques such as multiversion, but also that similarity detection tools may be useful in determining which versions to combine.

5.6.6 Contributions to the Monitoring Library

In this project, we had the opportunity to expand the "monitors" library created by our colleague Pedro Pinto in the *SPECS* group. The substitution policies *FIFO*, *LRU*, *LFU*, *MRU*, and *RANDOM* have been implemented. The user can choose which one to use for each variable or parameter to be monitored, enabling efficient management based on the user's preferences.

In addition to the substitution policies, two parameters were implemented that can count the total number of values assigned to the monitored variable that entered the hashtable and the total number of values substituted in the table. These parameters enable a more accurate comparison of the various substitution policies and their efficacy in table management.

It has also been made possible for the user to select a range of values within which the possible values assigned to the monitored variable must be located. This feature enables the user to establish more precise and specific data analysis criteria. This results in a more precise and direct understanding of the behaviour of the substitution policies within that specific range, and it may also contribute to savings in storage and processing computational resources.

5.7 Summary

This chapter presented the experimental setup and presented, analysed, and justified the results obtained by the proposed approach when applied to the benchmarks described in Section 3.8 of Chapter 3 and Chapter 4. Furthermore, the results of the study on the relationship between the resource usage resulting from the multiversion versions and the degree of similarity of their versions are also presented, as well as the results concerning the extension of the "monitors" library. Finally, an analysis of the code changes to be made for automation is carried out in Section 5.5 and a general analysis of the approach results is made in Section 5.6.

In order to ensure the reproducibility and openness of the experiments, the experimental setup is presented in Section 5.1. The investigation was carried out on a *LAPTOP-HTC0Q8HI* equipped with an *Intel(R) Core(TM) i7-8750H* processor and *16 GB* of *RAM*. Targeting the *Zynq-7000 SoC XC7Z020-1CLG400C FPGA*, optimized versions of the benchmarks were simulated and synthesised using the *Xilinx's Vitis HLS 2022.2* tool.

The results obtained from each optimized version were compared with the generic or original version of the respective benchmark, using metrics such as speedup, percentage of resources used, and area-delay product, allowing a comprehensive view of the performance, efficiency, and resource usage of the optimized versions of the approach.

The results obtained for the motivational example of the *pow* function and benchmarks *kNN* and *Disparity* are presented, analysed, and justified in Section 5.2. In summary, specialized variants of the *pow* function provide significant speed increases and a reduction in total resource usage when compared to the generic base version in specific scenarios. This can improve energy efficiency, reduce the resulting circuit area, and lower total expenditure. Compared to the respective original or generic version, the *multiversion* (*MV*) versions have increased or maintained specific accelerations. However, they consume less than half the resources of the sum of the related versions separately, which means a more efficient use of available resources, resulting in better performance and resource savings. It is essential to keep in mind that these improvements may differ based on the characteristics of the input data and the chosen optimization options.

Regarding the analysis of the relationship between the degree of similarity and performance in the multiversion technique, a metric capable of assessing the relative gain in terms of resource reduction that the multiversion version offers compared to the individual specialized versions was used. The results obtained so far indicate that the degree of similarity between the combined versions can potentially have an effect on resource utilisation in multiversion versions, but are insufficient to demonstrate a causal relationship between the two. However, it was found that even when combining more than two versions with a certain degree of similarity, we can have a significant effect on the resource use of multiversion versions. Moreover, the results indicate that resource utilisation was significantly reduced in all generated multiversion versions compared to the respective combined versions, in particular for the versions generated from the clusters suggested by the *AC* tool.

The extension of the "monitors" library explores different data management approaches and improves monitoring efficiency by making it more flexible and adaptable to different variable monitoring requirements and scenarios.

In Section 5.5, the main considerations and modifications to be made to the original code to enable the automation of the specialization process are discussed.

Finally, Section 5.6 concludes with an evaluation of the results obtained and a comparison with those reported in the state of the art. The main conclusions and results regarding the benefits and limitations of the proposed approach are discussed. The results obtained throughout the work were overall satisfactory and provided insight into the possible impact of applying techniques such as code specialization and multiversion for *FPGAs* using *HLS* tools. The study of the relationship between features and similarity highlighted the importance of proper selection of specialized versions to obtain an optimized multiversion. The contributions to the variable monitoring library allowed us to explore different monitoring approaches and adapt the tool to the specific needs of the project.

Chapter 6

Conclusion

This chapter presents an overview of the work performed and its outcomes. The main contributions are also discussed, as are potential future improvements.

6.1 Overview

Code specialization for high-level synthesis tools and *FPGAs* has been a complex and pertinent research topic, with the potential to considerably boost system performance and efficiency. The primary objective of this dissertation was to propose efficient code specialization techniques that would make code more suitable for synthesis in *HLS* tools and execution in *FPGAs*, resulting in significant enhancements. In some instances, multiversioning techniques were utilised to assure the existence of highly efficient accelerators in common scenarios and the ability to accelerate the application in other scenarios.

We identified existent gaps and unused capabilities in the context of code specialization for *HLS* and *FPGAs* through a comprehensive review of the state-of-the-art. This enabled us to comprehend the significance and limitations of current techniques. Despite the increase in research in this area, there is still an absence of techniques that exploit the potential of specialization techniques, particularly hardware-driven specialization. Based on this knowledge, we proposed a code specialization approach, which is described in Chapter 3 using a diagram that is divided into three main phases: information acquisition, application and analysis of techniques in an appropriate environment, and *FPGA* evaluation. The identification of this workflow enabled the determination of a series of implementation stages for our strategy and the evaluation of its efficacy. Following the phase of data collection, we researched techniques for capturing pertinent information about the behaviour of the selected benchmarks, *kNN* and Disparity, and identifying the critical regions of their codes. We utilised analysis and profiling tools such as *Gprof*, as well as techniques based on "Value profiling" methodologies, such as the use of variable monitoring tools like the "monitors" library. This enabled us to identify memory access patterns, data dependencies, and performance constraints that could be optimized. Analysing the benchmarks enabled the identification and optimization of crucial portions of the code, resulting in more efficient and scalable solutions that allowed for a more exhaustive analysis of certain algorithmic variables and intermediate values. In the techniques application phase, customised versions of each benchmark were developed based on the information obtained in the preceding phase. Our analysis enabled us to determine the most advantageous specializations and assess their effect on performance and resource consumption. Most of these transformations are static specializations of constants that supplant variable references with their corresponding constant values during compilation or code interpretation. There are, however, very specific and complex circumstances in which manual intervention was necessary. Metrics such as speedup, percentage of resources consumed, and *ADP* were used in comparison to generic implementations of each benchmark. These metrics facilitated the evaluation and analysis of results, enabling a quantitative evaluation of the performance, resource utilisation, and effectiveness of the specialized versions in comparison to the generic version. The results of the synthesis of these specializations, using the *Vitis HLS* tool to target board are highly positive, especially in cases where the specialized variables have a greater impact and constant presence or influence on the algorithms. These specializations were able to eliminate the need to recover the variable value at runtime, which may result in code that utilises its resources more efficiently and quickly. The multiversions whose synthesis results were analysed were statically generated from the combinations of the specialized version and the respective generic version of the benchmark and were selected at runtime based on an "opt" variable. Despite a slight increase in the percentage of total resource utilisation, the results were generally beneficial, as they indicate that the sum of the specialized and generic versions that correspond to the specialized versions of each multiversion has a percentage increase that is more than twice that of the multiversion versions while achieving nearly the same latency and resource utilisation as the generic version. Due to resource sharing and the elimination of overlaps, there was consequent resource optimization. The obtained results support the notion that multiversion is preferable in these circumstances, as it can combine the advantages of the generic and specialized versions while avoiding their disadvantages. It is essential to note, however, that these enhancements may vary depending on the case parameters, input data, and optimization options chosen.

Additionally, a study was conducted to determine the correlation between resource utilisation resulting from multiversion versions and the degree of similarity between their versions. Although preliminary results did not establish a clear causal relationship between these two factors, it was observed that the combination of versions with varying degrees of similarity can affect resource utilisation in multiversion versions, resulting in significant cost savings compared to combined versions.

Furthermore, the library of monitors was expanded by investigating various data management approaches and enhancing the monitoring's efficacy, making it more flexible and adaptable to different requirements and variable monitoring scenarios.

Although the results obtained have been encouraging, we recognise that there are still obstacles to overcome and possibilities for future research. Code-to-hardware specialization is an ever-evolving field with multiple potential paths of research. In conclusion, this dissertation presented an approach for code specialization using *HLS* tools and targeting *FPGAs* and demonstrated its potential to enhance the overall efficacy of systems. The results acquired support the practicability and applicability of this approach and suggest paths for future research and development. With continued progress in this area, it is anticipated that code specialization will become increasingly efficient and extensively adopted in hardware designs, driving significant advances in high performance computing.

6.2 Contributions

In this dissertation, we highlight the use of hardware accelerators, such as *FPGAs*, as an opportunity to increase performance and the need to address the challenges that their use presents. We expose existing code optimization options in this context, which include specialization and multiversion techniques, and highlight the scarcity of information on the use of code specialization in hardware designs. We were able to propose and describe an efficient approach for application code specialization in *HLL*, *C*, in order to make it more suitable for the use of synthesis tools in *HLS* and execution in *FPGAs*, resulting in significant performance enhancements. Multiversioning techniques were also employed to assure the existence of highly efficient accelerators in common scenarios as well as the application's ability to be accelerated in other scenarios. In this approach, we emphasise the significance of benchmark selection and analysis. We discussed the use of code analysis and profiling tools, as well as the "monitors" library for monitoring variables and intermediate values during program execution. In addition, a study was conducted on the correlation between the resource utilisation resulting from multiversion versions and the degree of similarity of their versions, which could be helpful in determining which versions should be chosen to generate multiversion versions. A library extension consisting of the implementation of substitution policies, parameters that count the number of values inserted in its hashtable as well as the number of substitutions in it, and an option to restrict the range of values that can enter the table was also added to the "monitors" library. These contributions made it possible to explore and compare various substitution policies, thereby expanding the library's functionalities and application scope, thereby making it more flexible and adaptable to various monitoring requirements and scenarios. In conclusion, this work contributes to the advancement of code specialization in hardware designs by exposing and exploring existing techniques and providing an effective approach to explore code specialization, seeking to fill the existing gap in the literature regarding code specialization in hardware, which has been widely studied in software designs and paving the way for the optimization of computing systems based on *HLS* and *FPGAs* and propelling the field forward.

6.3 Future work

While this work provided promising outcomes and demonstrated the viability of the proposed approach for code specialization using *HLS* tools and targeting *FPGAs*, there are still numerous opportunities for future research and enhancements. In this chapter we present some of these opportunities.

6.3.1 Automating Transformations for Specialized Version Development

The automation of the transformations required for specialized version development is an area of interest. While critical regions and opportunities for specialization have been identified, the process would benefit from exploring automated approaches that can traverse the source code, identify candidate regions, and efficiently implement the necessary transformations. This may necessitate the development of tools capable of analysing the code and modifying variables and parameters based on scenario-specific values.

6.3.2 Automating the Choice of the Versions composing the Multiversion Versions

The process of selecting the versions that would comprise the multiversion versions was one of the stages of this project. Although we used manual strategies and similarity detection tools between versions to facilitate the selection of which versions should comprise a multiversion version, there is still room for improvement and automation in this process. It would be interesting to investigate further whether there is a correlation between the degree of similarity between different specialized versions and the result of the number of resources for a version composed of them using the multiversion technique. Based on this research, scripts or tools capable of recommending parameter settings for the multiversion version could be developed, taking into account predefined criteria such as the degree of similarity between specialized versions. For instance, defining a threshold for the degree of similarity between versions to determine which of them would result in an effective multiversion would be one way to achieve this.

6.3.3 Using Benchmarks for Further Research

While the benchmarks chosen for this study were sufficient for evaluating performance and opportunities for specialization, there is room to expand the range of benchmarks and allow for additional research. In order to investigate a broader range of specializations, it would be advantageous to include benchmarks that represent various operations and situations. This could uncover additional patterns that could be automated and provide valuable insights for the approach's continued development.

6.3.4 Study on the Impact of Large-Scale Specializations of powf and sqrtf

A specific interesting study might be carried out to explore the impact of large-scale specializations of functions. For example, due to the difference in the precision of the computations, we observed that the use of the *powf/sqrtf* function as opposed to the *pow/sqrt* function resulted in improved performance and a more efficient use of resources. It would be interesting to research in depth the impact of this kind of specializations in various scenarios and identify situations in which the replacement of such functions can result in significant performance and efficiency improvements.

6.3.5 Study on the Suitable Tool for Assessing the Relationship between Similarity and Resource Utilization

To gain a more comprehensive understanding of the correlation between the degree of similarity between specialized versions and resource utilisation in multiversion versions, it is necessary to investigate more appropriate tools. Until now, we have measured similarity based on the entire code of the expert versions. Nonetheless, it is essential to consider intrinsic characteristics that may influence resource utilisation, such as memory access patterns, arithmetic operations, and multiplexer utilisation. It would be beneficial to develop a strategy that incorporates more granular metrics in order to gain a more comprehensive comprehension of the factors that influence the performance and efficiency of multiversion versions.

6.3.6 Impact Study of Multiversion Versions for Different Benchmark Scenarios

Lastly, an additional study could be conducted to assess the influence of multiversion versions for various benchmark scenarios. In the present work, we developed multiversion versions for specific benchmark scenarios, but it would be interesting to investigate multiversion versions that support multiple benchmark scenario types. This would enable for a more thorough analysis of the results after *HLS* synthesis and provide additional insight into the performance and efficacy of multiversion versions in various situations.

References

[1] Udaree Kanewala, Kesara Gamlath, Hasindu Ramanayake, Kalindu Herath, Isuru Nawinne, and Roshan Ragel. Power-aware high-level synthesis flow for mapping fpga designs. In *2019 Moratuwa Engineering Research Conference (MERCon)*, pages 228–233, 2019. doi:10.1109/MERCon.2019.8818883.

[2] Peter Flake, Phil Moorby, Steve Golson, Arturo Salz, and Simon J Davidmann. Verilog hdl and its ancestors and descendants. *Proc. ACM Program. Lang.*, 4(HOPL):87–1, 2020.

[3] Andrew Boutros and Vaughn Betz. Fpga architecture: Principles and progression. *IEEE Circuits and Systems Magazine*, 21(2):4–29, 2021. doi:10.1109/MCAS.2021.3071607.

[4] Clayton J. Faber, Steven D. Harris, Zhili Xiac, Roger D. Chamberlain, and Anthony M. Cabrera. Challenges designing for fpgas using high-level synthesis. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2022. doi:10.1109/HPEC55821.2022.9926398.

[5] Kaijie Wei, Koki Honda, and Hideharu Amano. Fpga design for autonomous vehicle driving using binarized neural networks. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 425–428, 2018. doi:10.1109/FPT.2018.00091.

[6] Vasileios Tsoutsouras, Konstantina Koliogeorgi, Sotirios Xydis, and Dimitrios Soudris. An exploration framework for efficient high-level synthesis of support vector machines: case study on ecg arrhythmia detection for xilinx zynq soc. *Journal of Signal Processing Systems*, 88(2):127–147, 2017.

[7] Mohammad Ehsanul Alim, Nazmus Sakib Bin Alam, and Sarosh Ahmad. Rfid based security and home automation system using fpga. In *2020 IEEE 7th International Conference on Engineering Technologies and Applied Sciences (ICETAS)*, pages 1–5, 2020. doi:10.1109/ICETAS51660.2020.9484179.

[8] Tsubasa Takaki, Yang Li, Kazuo Sakiyama, Shoei Nashimoto, Daisuke Suzuki, and Takeshi Sugawara. An optimized implementation of aes-gcm for fpga acceleration using high-level synthesis. In *2020 IEEE 9th Global Conference on Consumer Electronics (GCCE)*, pages 176–180, 2020. doi:10.1109/GCCE50665.2020.9291973.

[9] Sunil Puranik, Mahesh Barve, Dhaval Shah, Sharad Sinha, Rajendra Patrikar, and Swapnil Rodi. Key-value store using high level synthesis flow for securities trading system. In *2020 International Conference on Computing, Electronics Communications Engineering (iCCECE)*, pages 237–242, 2020. doi:10.1109/iCCECE49321.2020.9231158.

[10] Takeshi Ohkawa, Yuhei Sugata, Harumi Watanabe, Nobuhiko Ogura, Kanemitsu Ootsu, and Takashi Yokota. High level synthesis of ros protocol interpretation and communication circuit for fpga. In *2019 IEEE/ACM 2nd International Workshop on Robotics Software Engineering (RoSE)*, pages 33–36, 2019. doi:10.1109/RoSE.2019.00014.

[11] S.L. Bade and B.L. Hutchings. Fpga-based stochastic neural networks-implementation. In *Proceedings of IEEE Workshop on FPGA's for Custom Computing Machines*, pages 189–198, 1994. doi:10.1109/FPGA.1994.315612.

[12] Jintao Su, Hua Zhang, and Yunqi Yao. Design of anti-submarine warfare module for unmanned surface vehicle. In *2020 3rd International Conference on Unmanned Systems (ICUS)*, pages 342–345, 2020. doi:10.1109/ICUS50048.2020.9274874.

[13] M. Diaby, M. Tuna, J.L. Desbarbieux, and F. Wajsburt. High level synthesis methodology from c to fpga used for a network protocol communication. In *Proceedings. 15th IEEE International Workshop on Rapid System Prototyping, 2004.*, pages 103–108, 2004. doi:10.1109/IWRSP.2004.1311103.

[14] Majid Dashtbani, Amir Rajabzadeh, and Mohsen Asghari. High level synthesis as a service. In *2015 5th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 331–336, 2015. doi:10.1109/ICCKE.2015.7365851.

[15] Piotr Tomikowski and Gustaw Mazurek. Acceleration of radio direction finder algorithm in fpga computing platform. In *2022 23rd International Radar Symposium (IRS)*, pages 279–282, 2022. doi:10.23919/IRS54158.2022.9905066.

[16] Mingjun Jiao, Yue Li, Pengbo Dang, Wei Cao, and Lingli Wang. A high performance fpga-based accelerator design for end-to-end speaker recognition system. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 215–223, 2019. doi:10.1109/ICFPT47387.2019.00033.

[17] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design Test of Computers*, 26(4):8–17, 2009. doi:10.1109/MDT.2009.69.

[18] João M. P. Cardoso and Markus Weinhardt. *High-Level Synthesis*, pages 23–47. Springer International Publishing, Cham, 2016. URL: https://doi.org/10.1007/978-3-319-26408-0_2, doi:10.1007/978-3-319-26408-0_2.

[19] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016. doi:10.1109/TCAD.2015.2513673.

[20] Sakari Lahti, Panu Sjövall, Jarno Vanne, and Timo D. Hämäläinen. Are we there yet? a study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(5):898–911, 2019. doi:10.1109/TCAD.2018.2834439.

[21] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. Fpga hls today: Successes, challenges, and opportunities. *ACM Trans. Reconfigurable Technol. Syst.*, 15(4), aug 2022. URL: https://doi.org/10.1145/3530775, doi:10.1145/3530775.

[22] Philippe Coussy, Daniel D. Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design Test of Computers*, 26(4):8–17, 2009. doi:10.1109/MDT.2009.69.

[23] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. Accelerator-rich architectures: Opportunities and progresses. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014. doi:10.1145/2593069.2596667.

[24] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *2009 IEEE 31st International Conference on Software Engineering*, pages 287–297, 2009. doi:10.1109/ICSE.2009.5070529.

[25] Antoine Morvan, Steven Derrien, and Patrice Quinton. Polyhedral bubble insertion: A method to improve nested loop pipelining for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(3):339–352, 2013. doi:10.1109/TCAD.2012.2228270.

[26] Preeti Ranjan Panda, Namita Sharma, Srikanth Kurra, Khushboo Anil Bhartia, and Neeraj Kumar Singh. Exploration of loop unroll factors in high level synthesis. In *2018 31st International Conference on VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID)*, pages 465–466, 2018. doi:10.1109/VLSID.2018.115.

[27] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. From c/c++ code to high-performance dataflow circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(7):2142–2155, 2022. doi:10.1109/TCAD.2021.3105574.

[28] João Manuel Paiva Cardoso, José Gabriel de Figueired Coutinho, and Pedro C Diniz. *Embedded computing for high performance: Efficient mapping of computations using customization, code transformations and compilation*. Morgan Kaufmann, 2017.

[29] Qiang Liu, George A. Constantinides, Konstantinos Masselos, and Peter Y. K. Cheung. Combining data reuse with data-level parallelization for fpga-targeted hardware compilation: A geometric programming framework. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(3):305–315, 2009. doi:10.1109/TCAD.2009.2013541.

[30] Robert Muth, Scott Watterson, and Saumya Debray. *Code Specialization based on Value Profiles*, volume 1824. 6 2000.

[31] Minhaj Ahmad Khan, Henri-Pierre Charles, and Denis Barthou. Hybrid specialization: A trade-off between static and dynamic specialization. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 415–415, 2007. doi:10.1109/PACT.2007.4336243.

[32] Youenn Lebras, Andres S. Charif-Rubial, and William Jalby. Combining static and dynamic analysis to guide pgo for hpc applications: a case study on real-world applications. In *2019 International Conference on High Performance Computing Simulation (HPCS)*, pages 513–520, 2019. doi:10.1109/HPCS48598.2019.9188161.

[33] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 259–269. IEEE, 1997.

[34] Yoshihiko Futamura. Partial Computation of Programs. -1 1999.

[35] F. Noel, L. Hornof, C. Consel, and J.L. Lawall. Automatic, template-based run-time specialization: implementation and experimental study. In *Proceedings of the 1998 International Conference on Computer Languages (Cat. No.98CB36225)*, pages 132–142, 1998. doi:10.1109/ICCL.1998.674164.

[36] Jason Lau, Aishwarya Sivaraman, Qian Zhang, Muhammad Ali Gulzar, Jason Cong, and Miryung Kim. Heterorefactor: Refactoring for heterogeneous computing with fpga. pages 493–505, 2020. doi:https://doi.org/10.1145/3377811.3380340.

[37] Dan Quinlan and Chunhua Liao. The rose source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, volume 2011, page 1. Citeseer, 2011.

[38] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[39] Xilinx Virtex UltraScale+ FPGA VCU1525 Acceleration Development Kit. https://www.xilinx.com/products/boards-and-kits/vcu1525-a.html. [Online; accessed 2022-12-26].

[40] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE, 2014.

[41] Vivado ML Overview. https://www.xilinx.com/products/design-tools/vivado.html. [Online; accessed 2022-12-26].

[42] M. Akif Özkan, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, Roland Leißa, Sebastian Hack, Jürgen Teich, and Frank Hannig. Anyhls: High-level synthesis with partial evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3202–3214, 2020. doi:10.1109/TCAD.2020.3012172.

[43] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. Anydsl: A partial evaluation framework for programming high-performance libraries. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018.

[44] Roland Leißa, Klaas Boesche, Sebastian Hack, Richard Membarth, and Philipp Slusallek. Shallow embedding of dsls via online partial evaluation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 11–20, 2015.

[45] FPGA Intel. Opencl best practices guide. *URL http://www. altera. com/en_US/pdfs/literature/hb/opencl-sdk/aocl-bestpractices-guide. pdf*, 2017.

[46] Cyclone® V GT FPGA - Intel® FPGA. https://www.intel.com/content/www/us/en/products/details/fpga/cyclone/v/gt.html. [Online; accessed 2022-12-26].

[47] Xilinx Zynq-7000 SoC ZC702 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html. [Online; accessed 2022-12-26].

[48] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[49] Oliver Reiche, M Akif Özkan, Richard Membarth, Jürgen Teich, and Frank Hannig. Generating fpga-based image processing accelerators with hipacc. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1026–1033. IEEE, 2017.

[50] Yilin Dong, Rigui Zhou, Changming Zhu, Lei Cao, and Xianghui Li. Hierarchical activity recognition based on belief functions theory in body sensor networks. *IEEE Sensors Journal*, 22(15):15211–15221, 2022. doi:10.1109/JSEN.2022.3186086.

[51] Zhun-Ga Liu, Lin-Qing Huang, Kuang Zhou, and Thierry Denœux. Combination of transferable classification with multisource domain adaptation based on evidential reasoning. *IEEE Transactions on Neural Networks and Learning Systems*, 32(5):2015–2029, 2021. doi:10.1109/TNNLS.2020.2995862.

[52] Zhunga Liu, Xuxia Zhang, Jiawei Niu, and Jean Dezert. Combination of classifiers with different frames of discernment based on belief functions. *IEEE Transactions on Fuzzy Systems*, 29(7):1764–1774, 2021. doi:10.1109/TFUZZ.2020.2985332.

[53] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. doi:10.1162/neco.1997.9.8.1735.

[54] Fionn Murtagh and Pierre Legendre. Ward's hierarchical clustering method: clustering criterion and agglomerative algorithm. *arXiv preprint arXiv:1111.6285*, 2011.

[55] Zhenghua Chen, Chaoyang Jiang, and Lihua Xie. A novel ensemble elm for human activity recognition using smartphone sensors. *IEEE Transactions on Industrial Informatics*, 15(5):2691–2699, 2019. doi:10.1109/TII.2018.2869843.

[56] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra Perez, and Jorge Luis Reyes Ortiz. A public domain dataset for human activity recognition using smartphones. In *Proceedings of the 21th international European symposium on artificial neural networks, computational intelligence and machine learning*, pages 437–442, 2013.

[57] Oresti Banos, Rafael Garcia, Juan A Holgado-Terriza, Miguel Damas, Hector Pomares, Ignacio Rojas, Alejandro Saez, and Claudia Villalonga. mhealthdroid: a novel framework for agile development of mobile health applications. In *International workshop on ambient assisted living*, pages 91–98. Springer, 2014.

[58] William R Cook and Ralf Lämmel. Tutorial on online partial evaluation. *arXiv preprint arXiv:1109.0781*, 2011.

[59] Raquel Lazcano, Daniel Madroñal, Eduardo Juarez, and Philippe Clauss. Runtime multiversioning and specialization inside a memoized speculative loop optimizer. In *Proceedings of the 29th International Conference on Compiler Construction*, pages 96–107, 2020.

[60] Juan Manuel Martinez Caamaño, Manuel Selva, Philippe Clauss, Artyom Baloian, and Willy Wolff. Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones. *Concurrency and Computation: Practice and Experience*, 29(15):e4192, 2017.

[61] Aravind Sukumaran-Rajam and Philippe Clauss. The polyhedral model of nonlinear loops. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):1–27, 2015.

[62] Lianjie Luo, Yang Chen, Chengyong Wu, Shun Long, and Grigori Fursin. Finding representative sets of optimizations for adaptive multiversioning applications. *arXiv preprint arXiv:1407.4075*, 2014.

[63] https://open64-ici.sourceforge.net/. https://open64-ici.sourceforge.net/. [Online; accessed 2022-12-26].

[64] Konstantinos Koukos, Per Ekemark, Georgios Zacharopoulos, Vasileios Spiliopoulos, Stefanos Kaxiras, and Alexandra Jimborean. Multiversioned decoupled access-execute: The key to energy-efficient compilation of general-purpose programs. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 121–131, 2016.

[65] Alexandra Jimborean, Konstantinos Koukos, Vasileios Spiliopoulos, David Black-Schaffer, and Stefanos Kaxiras. Fix the code. don't tweak the hardware: A new compiler approach to voltage-frequency scaling. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 262–272, 2014.

[66] Mark Stephenson, Ram Rangan, and Stephen W Keckler. Cooperative profile guided optimizations. In *Computer Graphics Forum*, volume 40, pages 71–83. Wiley Online Library, 2021.

[67] Georgios Zacharopoulos. *Compiler Analysis for Hardware/Software Co-design and Optimization*. PhD thesis, 01 2020. doi:10.13140/RG.2.2.23474.96966/1.

[68] XILINX. Vitis high-level synthesis user guide (ug1399). In *Documentation Portal*, 2021.

[69] Tul. https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html. [Online; accessed 2022].

[70] Xup PYNQ-Z2. https://www.xilinx.com/support/university/xup-boards/XUPPYNQ-Z2.html. [Online; accessed 2022].

[71] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. Sd-vbs: The san diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64, 2009. doi:10.1109/IISWC.2009.5306794.

[72] Tiagolascasas, mar 2023.

[73] GNU Project. *GNU gprof*. Free Software Foundation, 2023. URL: https://sourceware.org/binutils/docs/gprof/.

[74] GNU Project. *GNU g2prof*. Free Software Foundation, 2023. URL: https://sourceware.org/binutils/docs/gprof/g2prof.html.

[75] Mark Stephenson and Ram Rangan. Pgz: automatic zero-value code specialization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 36–46, 2021.

[76] Adriano Sanches, João MP Cardoso, and Alexandre CB Delbem. Identifying merge-beneficial software kernels for hardware implementation. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 74–79. IEEE, 2011.

[77] Manuel Freire. manuel-freire/ac2. https://github.com/manuel-freire/ac2, 2023.

[78] Xilinx Inc. *Vivado Design Suite User Guide*, Setembro 2022. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2022_2/ug973-vivado-release-notes-install-license.pdf.

[79] XILINX. Python productivity for zynq (pynq) documentation. volume 2.5, pages 3–20, 2020.

[80] Ioannis Stamoulias and Elias S Manolakos. Parallel architectures for the knn classifier–design of soft ip cores and fpga implementations. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):1–21, 2013.

[81] Mokhles Aamel Mohsin. *An FPGA-based hardware accelerator for k-nearest neighbor classification for machine learning*. University of Colorado Colorado Springs, 2017.

[82] Xilinx. *Vitis HLS User Guide*. Xilinx Inc., 2021. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug902-vitis-high-level-synthesis.pdf.