WASHINGTON UNIVERSITY IN ST. LOUIS

McKelvey School of Engineering
Department of Electrical & Systems Engineering

Thesis Examination Committee:
Roger Chamberlain, Chair
James Buckley
Shantanu Chakrabartty

Enhancing FPGA Synthesis for Space Applications: Performance Evaluation of ScaleHLS in the ADAPT Project

by
Ruoxi Wang

A thesis presented to
the McKelvey School of Engineering
of Washington University in
partial fulfillment of the
requirements for the degree
of Master of Science

May 2024
St. Louis, Missouri

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am profoundly thankful to my advisor, Professor Roger Chamberlain, for his continuous support and guidance throughout my research journey. My gratitude also extends to Professor James Buckley and Professor Shantanu Chakrabartty for their insightful contributions and for joining my committee.

Special acknowledgment is due to the team at the Stream-Based Supercomputing Lab, especially Marion Sudvarg, whose assistance was vital in navigating the complexities of coding. I am equally grateful to Izabella Pastrana for her expert advice on FPGA-related issues, and to Anthony Cabrera, whose innovative ideas have significantly enhanced my work.

I deeply appreciate all my friends who stood by me during my defense, and those who provided mental support during challenging times with coding and writing. Listing each of you would indeed fill an entire page.

I must also express my profound gratitude to my parents, whose financial support enabled me to pursue my academic interests at Washington University in St. Louis and explore new opportunities in technology.

As I conclude my master's journey, I extend a special thanks to myself for persevering and maintaining my resolve. This journey has been one of immense growth and self-discovery, reminding me that the pursuit of knowledge is a never-ending journey that stretches beyond the horizons of graduation.

Lastly, I would like to acknowledge the assistance of ChatGPT, an invaluable tool that helped refine my work and clarify my thoughts throughout the writing process.

To all who have walked this path with me: best wishes, and may we all be better versions of ourselves the next time we meet.

Ruoxi Wang

Washington University in St. Louis
May 2024

ABSTRACT OF THE THESIS

Enhancing FPGA Synthesis for Space Applications:

Performance Evaluation of ScaleHLS in the ADAPT Project

by

Ruoxi Wang

Master of Science in Electrical Engineering

Washington University in St. Louis, 2024

Professor Roger Chamberlain, Chair

This thesis investigates the application of ScaleHLS, a high-level synthesis (HLS) tool, to enhance Field-Programmable Gate Array (FPGA) synthesis for space applications, with a focus on the Antarctic Demonstrator Advanced Particle-astrophysics Telescope (ADAPT) project. The study explores how ScaleHLS optimizes the transformation of C code into FPGA-compatible designs to improve computational efficiency and resource utilization.

The research details the process of adapting ADAPT's computational algorithms for FPGA using ScaleHLS, emphasizing the tool's effectiveness in streamlining the code-to-hardware translation. A performance evaluation highlights significant improvements in resource management and operational speed, demonstrating the tool's impact on FPGA synthesis.

These findings illustrate ScaleHLS's potential to advance hardware design and set a foundation for further developments in high-level synthesis, crucial for enhancing spaceborne computational systems.

Chapter 1

Introduction

The Advanced Particle-astrophysics Telescope (APT) project seeks to benefit space-based observatories through the integration of cutting-edge FPGA technology, addressing the critical balance between computational speed and resource efficiency required in space applications. Historically, while FPGAs have been the go-to solution for processing data at high speeds with limited onboard resources, traditional high-level synthesis (HLS) tools have struggled with efficiently translating C code into HDL, often resulting in suboptimal performance and resource utilization. This thesis explores how the innovative HLS tool, ScaleHLS, emerges as a potentially pivotal development in this area, enhancing our ability to handle complex computational challenges in astrophysics by optimizing the translation process, thereby improving data processing efficiency and reducing resource consumption essential for the demanding environment of space.

1.1 APT and ADAPT

1.1.1 Advanced Particle-astrophysics Telescope Project

The Advanced Particle-astrophysics Telescope (APT) is designed to advance our understanding of cosmic phenomena, specifically targeting dark matter and the mechanics of neutron-star mergers. An image illustrating APT is shown in Figure 1.1. At the core of APT's operation, Field-Programmable Gate Arrays (FPGAs) are employed throughout the front-end readout electronics to control data acquisition from sensors and associated analog-to-digital converter electronics. By leveraging these resources for computation, FPGAs facilitate the initial stages of the data processing pipeline, where High-Level Synthesis (HLS) plays a crucial role in handling the complex demands of astrophysical research. These

technologies enable APT to process data with exceptional efficiency and flexibility, making real-time analysis and localization of cosmic events feasible [2].

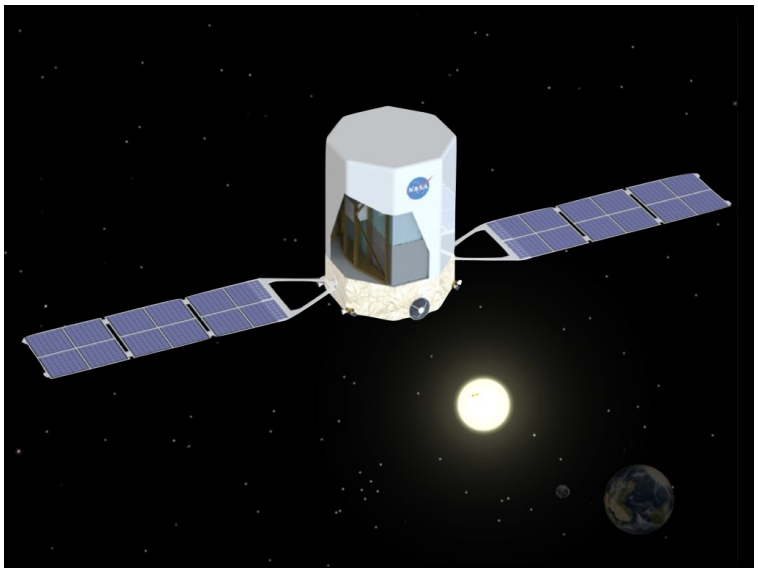

Figure 1.1: A rendering of the APT instrument [1].

APT combines a pair tracker and a Compton telescope into a unified platform, designed to be optimized for a broad energy spectrum. The integration of FPGA and HLS technologies underpins a dynamic and efficient computational framework, crucial for the front-end processing operations of APT. FPGAs offer a versatile hardware environment that can be reconfigured for various computational tasks in cosmic-ray detection, while HLS significantly streamlines the development process, enabling swift adaptations to emerging scientific discoveries [16].

This innovative integration not only boosts the telescope's data processing capacities but also establishes new standards in astrophysical research, allowing APT to perform highly sensitive observations over a diverse range of energies. The strategic deployment of FPGA and HLS technologies is vital to APT's mission, positioning it as a key instrument in exploring and understanding high-energy cosmic phenomena [15].

1.1.2 Antarctic Demonstrator for the APT

The Antarctic Demonstrator for the APT (ADAPT) serves as a critical pilot project, testing key technologies and methodologies under similar conditions to space flight through a long duration high-altitude stratopheric balloon flight. ADAPT is specifically designed to validate the operational capabilities and durability of the APT's instrumentation to advance the technical readiness level of the enabling technologies for APT. These insights are crucial for refining APT's design and enhancing its resilience, ensuring that the main telescope can withstand and operate effectively in the varied and challenging conditions encountered in space.

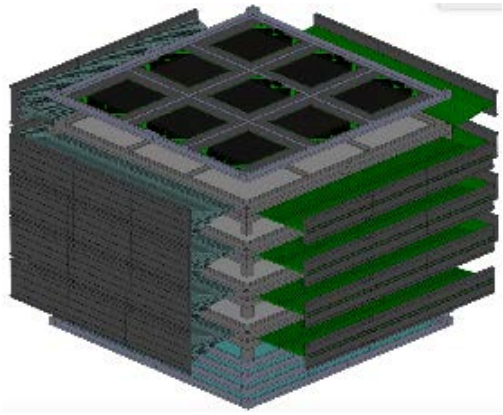

Figure 1.2: A detailed view of the ADAPT detector stackup [1].

As depicted in Figure 1.2, ADAPT employs a 4-layer stackup of detectors and electronics. Each layer includes 3×3 tiles of CsI:Na scintillator, which detect gamma rays whose direction of origin is to be determined by the instrument.

The placement of ADAPT in Antarctica allows researchers to replicate and study the effects of space-like isolation and extremities on the equipment, making it an invaluable proving ground for the technologies to be utilized in the APT. The data harvested from ADAPT is instrumental not only in diagnosing and perfecting hardware configurations but also in assessing the performance of HLS algorithms under real-world, challenging conditions. This preparatory stage is vital for the success of the APT, laying a solid groundwork for the ultimate deployment and operation of the telescope.

1.2 FPGA Role in APT

Field-Programmable Gate Arrays (FPGAs) play a pivotal role in the APT project, fulfilling the need for computational flexibility and efficiency in the processing of astrophysical data. The decision to employ FPGAs aboard the APT is underlined by their capability for high-speed data processing and their adaptability to the fluctuating computational demands inherent in space observation tasks [9, 17].

The advantages of using FPGAs in space applications are numerous:

- FPGAs' post-manufacturing reconfigurability enables adaptation of hardware circuits to specific tasks and algorithms, a critical feature in space where observational strategies evolve with scientific discoveries [7, 10].
- The in-situ reprogramming ability of FPGAs provides a practical solution for space missions, allowing for on-the-fly adjustments to new scientific challenges or mission parameters [12].
- Inherent suitability for parallel processing makes FPGAs ideal for managing large datasets and complex computations, pivotal for astrophysical research conducted in space [8].
- Enhanced resistance to radiation with specific design techniques and error-mitigation strategies equips FPGAs to handle the harsh space environment, protecting the integrity of data processing [10, 17].
- Lower power requirements of FPGAs align with the energy constraints of spacecraft, enabling more computations per watt and ensuring energy-efficient operations [7, 9].

These points illustrate why FPGAs are not just technical components but strategic elements in space exploration [8, 12].

1.3 HLS and ScaleHLS

High-level synthesis (HLS) tools are a relative newcomer to the field of digital design, providing a bridge between software programs and hardware implementation. These tools allow for high-level programming languages such as C or C++ to be synthesized into code that can be executed by FPGAs. Nevertheless, traditional HLS approaches often entail a labor-intensive optimization process, where developers are required to embed specific directives, like the #pragma directives, to guide the partitioning of arrays and the pipelining of processes with precise initiation intervals [3]. This manual optimization is not only time-consuming but also susceptible to human error, which can result in less than optimal FPGA utilization.

The challenge of achieving efficient high-level code translation into hardware descriptions is compounded by the intricacies of FPGA architectures, where a less-than-ideal synthesis can lead to a significant underutilization of the available resources. Recognizing these limitations, the development of innovative tools such as ScaleHLS has become imperative. ScaleHLS endeavors to automate and refine the synthesis process, thereby bolstering the adaptability and efficiency of FPGA usage in essential applications, including the ambitious APT project [3].

ScaleHLS represents an advancement in HLS practices by employing Multi-Level Intermediate Representation (MLIR) to enhance optimization across multiple levels of abstraction, from high-level architectural decisions to low-level synthesis details. Unlike traditional HLS tools confined to single-level abstraction, ScaleHLS operates on a MLIR to enhance design space exploration and optimization across various abstraction levels [20].

The essence of ScaleHLS lies in its hierarchical approach, utilizing the MLIR structure [6] to perform targeted optimizations across various abstraction levels, thus improving the synthesis quality and performance. An automated DSE engine, powered by dynamic programming and evolutionary algorithms, facilitates rapid evaluation and selection of design configurations, significantly boosting performance gains on FPGAs [18]. The ScaleHLS framework is illustrated in Figure 1.3. It includes 3 distinct abstraction levels of intermediate representation (IR): graph-level, loop-level, and directive-level. The usage in this thesis is illustrated in the far left of the figure, originating with C/C++ source code and culminating with C/C++ code designed for the manufacturer's HLS compiler.

ScaleHLS's robust transformation and analysis library encapsulates optimization tasks into modular components, offering flexibility and speeding up the HLS process. This modularity ensures that ScaleHLS not only meets the immediate needs of hardware designers but also adapts to evolving design requirements, making it an indispensable tool for advancing FPGA-based system designs and supporting projects like APT.

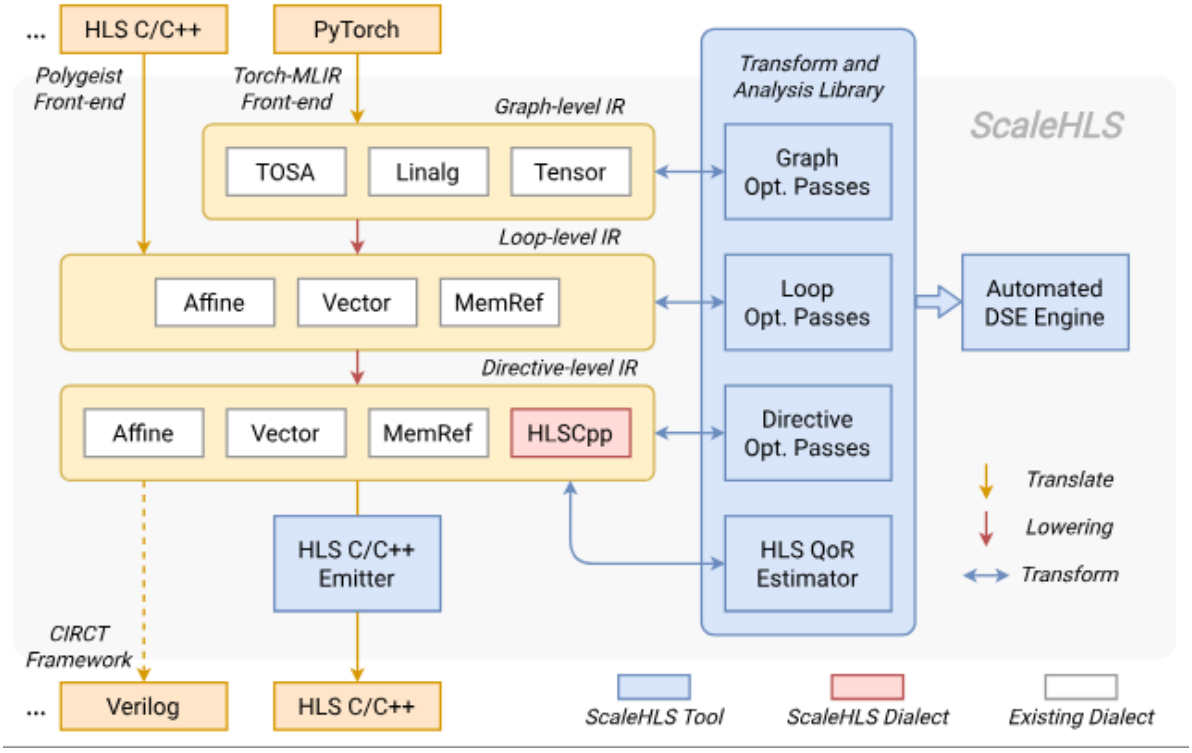

Figure 1.3: ScaleHLS framework [18].

1.4 Thesis Structure

This thesis is structured to provide an examination of ScaleHLS's capabilities and its application in FPGA design, with a particular focus on data preprocessing for the ADAPT project. Each chapter is crafted to progressively build upon the last, creating a cohesive narrative that underscores the contributions of this research. The structure is as follows:

- **Chapter 1: Introduction** – Introduces the research context, objectives, and the significance of the study. It provides an overview of the ADAPT project and outlines the potential contributions of ScaleHLS to FPGA-based design processes.

- **Chapter 2: Background** – Presents a literature review on HLS and its development, situating ScaleHLS within the broader context of HLS tools. It includes a discussion on early HLS tools, advancements in HLS technologies, and recent innovations that have shaped current practices.

- **Chapter 3: ScaleHLS Implementation** – Describes the methodology of applying ScaleHLS to the ADAPT project. It delves into the practical aspects of the implementation, discussing the challenges, solutions, and the specific optimizations made with ScaleHLS.

- **Chapter 4: Performance Evaluation and Results** – This chapter analyzes the improvements in FPGA performance using ScaleHLS for several algorithms within the ADAPT project. It contrasts the enhancements in resource utilization and processing speed with traditional HLS methods, showcasing the benefits and challenges of ScaleHLS.

- **Chapter 5: Conclusion and Future Work** – The conclusion summarizes the thesis' contributions to FPGA synthesis for space applications, highlighting the efficiencies gained through ScaleHLS. It also discusses potential future research directions to further enhance high-level synthesis tools.

This structure is designed to guide the reader through a logical progression from foundational concepts to practical application, culminating in a critical evaluation of the results. It ensures that the thesis is both informative and engaging, providing a clear narrative that demonstrates the value of ScaleHLS in FPGA design.

Chapter 2

Background and Related Work

2.1 Compton Computational Pipeline

The Compton computational pipeline is crafted to handle sensor data for astrophysical observations with high efficiency. It comprises a sequence of computational tasks that are executed by the FPGA-based front-end electronics, integral to preprocessing the sensor data into scientifically valuable information. The pipeline's key stages are illustrated in Figure 2.1.

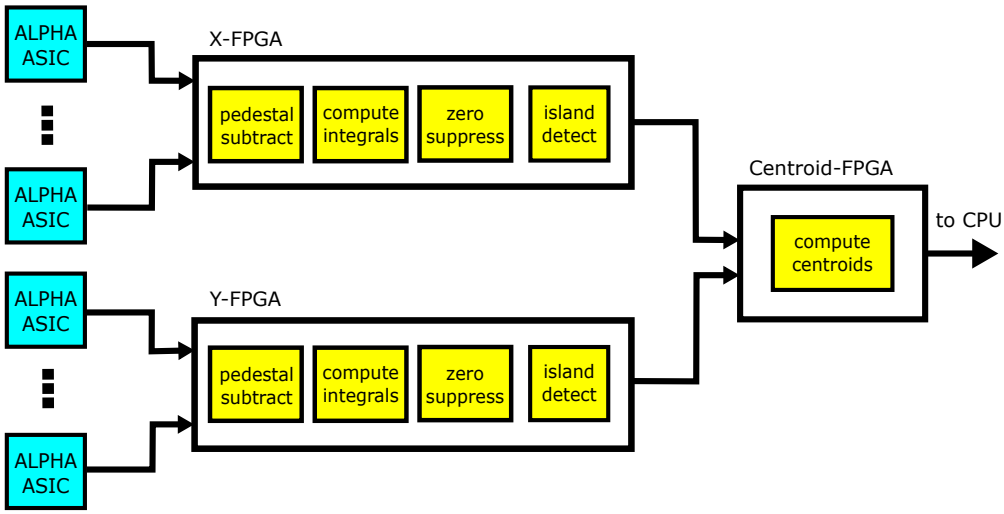

Figure 2.1: FPGA Algorithms of ADAPT [16].

There are five stages for the data process:

1. Pedestal Subtraction: The first step in the pipeline involves removing the baseline noise from the data, known as pedestal subtraction. This process is critical for isolating the true signal from the background noise inherent in the analog memory cells of the waveform

digitizers (ALPHA ASICs). The pedestal values are subtracted from the digitized readouts to obtain the actual signal values.

2. Signal Integration: Following pedestal subtraction, the signal integration stage sums up the signal over a specified window to quantify the energy captured by each pixel. This process involves integrating the waveform samples to infer the total number of photons detected, which is fundamental for determining the characteristics of the observed cosmic events.

3. Zero Suppression: To optimize data handling and storage, zero suppression is applied, which sets negligible signal values to zero. This stage reduces the volume of data that needs to be processed and transmitted, focusing on significant signal values that indicate actual astrophysical events.

4. Island Detection: The processed signals are then analyzed to identify clusters of adjacent pixels with non-zero values, termed *islands*. These islands represent potential astrophysical events or interactions within the detector. The identification of these islands is a crucial step in mapping the spatial distribution of the detected events.

5. Centroiding: The final stage in the pipeline is centroiding, where the center of gravity of the signal distribution within an island is calculated. This step provides a precise localization of the events, which is essential for reconstructing the direction and energy of the incident cosmic rays or gamma rays.

Implemented through High-Level Synthesis HLS techniques, each step is optimized for FPGA execution, enabling swift development cycles and effective hardware deployment [16]. The HLS method allows these complex computational tasks to be translated from high-level programming constructs into hardware instruction sets, finely tuning the pipeline for optimal speed, power economy, and resource management. Such an approach is indispensable for space instruments like ADAPT, where constraints on power and hardware are stringent and rapid, high-volume data processing is paramount.

This thesis delves into the integrated performance and efficiency of the aforementioned algorithms within the ADAPT computational pipeline, underscoring the significance of HLS in enhancing data processing for space-borne astrophysical observatories. Specifically, we

investigate the ability of ScaleHLS to eliminate the manual tuning required as part of the initial implementation by Sudvarg et al. [16].

2.2 Related Work

2.2.1 Early Developments in HLS

The journey of HLS began with innovative efforts to simplify hardware design processes, allowing software-centric approaches to FPGA programming. Early HLS initiatives sought to abstract complex hardware specifics, enabling software developers to utilize the parallel processing power of FPGAs effectively.

A landmark in early HLS development was Streams-C, introduced by Gokhale et al., which presented a stream-based programming model for FPGAs. This model facilitated the development of applications where data could be processed in parallel streams, significantly optimizing performance and resource utilization in FPGA designs [5].

Streams-C, as one of the early HLS tools, was pivotal in demonstrating that high-level programming concepts could be effectively mapped onto hardware architectures, thereby reducing development time and making FPGA technology more accessible to software engineers. This period marked a transformative phase in digital design, where the focus shifted towards higher abstraction levels, setting a precedent for the advanced HLS tools that followed.

These early developments in HLS were instrumental in laying the foundational principles for subsequent advancements in the field, highlighting the potential of high-level synthesis to bridge the gap between software programming and hardware implementation [4].

2.2.2 Advancements in HLS Tools

The evolution of HLS tools represents a significant phase in digital design, characterized by the transition from basic HLS concepts to sophisticated, integrated synthesis environments. This period witnessed the emergence of tools that not only simplified the translation

from high-level code to hardware designs but also enhanced the efficiency, performance, and scalability of these designs.

A notable advancement in HLS was the development of AutoPilot by Zhang et al., which later evolved into the Xilinx Vivado HLS tool. AutoPilot was among the first to provide a comprehensive environment for HLS, offering features like automated resource management, optimization, and the ability to target various FPGA platforms. This tool marked a shift towards more accessible and efficient FPGA programming, significantly impacting the development of subsequent HLS tools [21].

The progression in HLS tools has been marked by a continuous effort to improve the user experience and the quality of the synthesized hardware. Tools like the AMD Xilinx HLS suite, derived from the foundational concepts of AutoPilot, have integrated advanced optimization algorithms, support for a wider range of high-level programming languages, and more intuitive user interfaces. These improvements have democratized FPGA programming, enabling a broader range of developers, including those with limited hardware design expertise, to effectively utilize FPGA technology for various applications [3].

Moreover, the advancements in HLS have been paralleled by an increasing emphasis on design space exploration (DSE) and automated optimization. These features allow designers to navigate the complex landscape of hardware configurations more efficiently, identifying optimal solutions that balance performance, resource utilization, and power consumption. The integration of these capabilities into HLS tools has been crucial in managing the growing complexity of digital systems and meeting the stringent requirements of modern hardware designs [3].

In summary, the advancements in HLS tools over the years have significantly transformed the landscape of digital hardware design. By providing more sophisticated, user-friendly, and efficient tools, HLS has become an indispensable part of the hardware design process, enabling the rapid and effective development of complex digital systems.

2.2.3 Recent Innovations and Tool Improvements in HLS

In recent years, HLS has experienced a wave of innovations and tool improvements that have further enhanced its efficiency and application range. These advancements are characterized by the integration of machine learning algorithms, automated design space exploration (DSE), and increased support for complex design tasks, reflecting the dynamic nature of modern hardware design requirements.

The introduction of tools like AutoDSE by Sohrabizadeh et al. represents a significant leap forward in automating the HLS process. AutoDSE optimizes the design of FPGA accelerators by systematically navigating the design space to identify optimal configurations. This approach not only simplifies the design process but also significantly improves the performance and efficiency of the resulting hardware designs, making HLS more accessible and effective for a broader range of applications [14].

Empirical research, such as the work by Sanaullah et al., has provided valuable insights into optimizing HLS for specific hardware platforms like FPGA OpenCL. These studies have led to more targeted and effective optimization strategies, enhancing the performance of HLS-generated designs and reducing the gap between manually optimized hardware and HLS outputs [13].

Moreover, the integration of predictable models in HLS, as explored by Nigam et al., introduces a new dimension of reliability and predictability in accelerator design. This research addresses the challenge of designing time-sensitive systems, ensuring that HLS tools can produce hardware accelerators that meet stringent performance and timing requirements, crucial for applications in domains like real-time processing and embedded systems [11].

Recent innovations have also focused on improving the user experience in HLS tools, with developments aimed at making these tools more intuitive and easier to use for software programmers. This user-centric approach in tool development underscores the importance of HLS in bridging the gap between software development and hardware implementation, facilitating the adoption of FPGA technology across a wider range of industries and application areas.

In conclusion, the recent advancements in HLS technology and tools have significantly expanded the capabilities and reach of HLS in the hardware design process. By making HLS

more automated, efficient, and user-friendly, these innovations are paving the way for a new era in digital hardware design, where complex systems can be synthesized more rapidly and effectively than ever before.

Chapter 3

ScaleHLS Implementation

3.1 Introduction to ScaleHLS in the Context of ADAPT

ScaleHLS, an innovative High-Level Synthesis framework, has been designed to optimize the FPGA development process by automating the translation of high-level programming constructs into efficient hardware designs. The utilization of ScaleHLS promises to not only reduce the development time but also enhance the computational efficiency and efficacy of data processing within the ADAPT pipeline. This section will elucidate the role of ScaleHLS in the ADAPT project, assessing its ability to streamline the FPGA design cycle and exploring its impact on resource optimization and processing latency.

The subsequent analysis will delve into the methodologies employed to adapt the naive C preprocessing model to an FPGA-ready design using ScaleHLS, evaluating the effectiveness of ScaleHLS in automating pragma insertion, and examining its capacity for optimizing resource allocation. Through this exploration, we will answer key research questions posed in Chapter 2, documenting the pragmatic advantages and any limitations encountered in applying ScaleHLS to a complex, real-world computational task [16, 20]. The insights garnered from this inquiry will substantiate the feasibility and potential of employing ScaleHLS in high-stakes, performance-critical applications, setting a precedent for future HLS endeavors in spaceborne instruments.

3.2 ScaleHLS Methodology

This section delves into the detailed methodology employed in applying ScaleHLS for efficient FPGA synthesis, predominantly illustrated through the `ped_subtract` function as a

principal example. This function exemplifies the typical challenges and solutions in adapting ADAPT project codes to leverage ScaleHLS's automated synthesis capabilities. While `ped_subtract` serves as the focal point for demonstrating the transformation process, the methodologies and optimization strategies discussed herein are systematically applied to various other computational elements within the ADAPT project. This consistent application showcases the versatility and adaptability of ScaleHLS across different coding constructs and data processing requirements, underpinning its value in streamlining FPGA development in astrophysical instrumentation.

3.2.1 Preparation of Initial C Code for ScaleHLS

In the context of the ADAPT project, the `ped_subtract` function plays a crucial role in processing data from the detector and serves as a testbed for studying the suitability of ScaleHLS for subsequent analysis stages. It subtracts the pedestal value from each channel to normalize the signal, preparing it for further processing stages. The initial code is straightforward and written in C, designed to be compatible with ScaleHLS for subsequent optimizations and transformations.

Here is the naive implementation of the `ped_subtract` function, as found in the original codebase:

Listing 3.1: Naive Implementation of Pedestal Subtraction

```
void ped_subtract(Packet *pkt, unsigned *peds, unsigned a) {
    for (unsigned s = 0; s < NUM_SAMPLES; ++s) {
        unsigned idx = (pkt->starting_sample + s) % NUM_SAMPLES;
        for (unsigned c = 0; c < NUM_CHANNELS; ++c) {
            unsigned ped_idx = pkt->bank * NUM_SAMPLES * NUM_CHANNELS +
                               idx * NUM_CHANNELS + c;
            results[a][s][c] = pkt->samples[s][c] - peds[ped_idx];
        }
    }
}
```

The code iterates over samples and channels within each data packet, applying the pedestal subtraction to each data point. This initial version is essential for understanding the baseline performance and serves as the starting point for further optimizations through ScaleHLS [16].

3.2.2 Transformation to ScaleHLS-Compatible Code

Adapting the initial C code for compatibility with ScaleHLS required several transformations to meet the tool's syntactic and structural requirements, ensuring effective hardware synthesis. The primary modifications to the `ped_subtract` function and related code sections included:

1. **Converting C++ Constructs to C:** ScaleHLS specifically processes C code, necessitating the conversion of any C++ constructs into plain C. This transformation was crucial for ensuring that ScaleHLS could effectively analyze and optimize the code.

2. **Structures to Variables:** The original code utilized structures extensively for data organization. Given that ScaleHLS optimizes individual variables more efficiently than data structures, these structures were decomposed into separate scalar variables to simplify the hardware translation process.

3. **Refactoring Variable Reassignments:** The optimization challenges for ScaleHLS often stemmed from variables being reassigned within loops, which can complicate loop transformations. For example:

   Listing 3.2: Original Variable Reassignment in ScaleHLS

   ```
   unsigned idx = (starting_sample_number + i) % NUM_SAMPLES;
   ```

   To improve optimization, such reassignments were refactored to separate the steps involved, thereby simplifying the expressions and enhancing the clarity of operations:

   Listing 3.3: Refactored Variable Reassignments for ScaleHLS

   ```
   unsigned idx = starting_sample_number + i;
   unsigned idx1 = idx % NUM_SAMPLES;
   ```

   This modification aids ScaleHLS in more effectively analyzing and optimizing the code, particularly benefiting loop unrolling and pipeline configurations by clarifying dependencies and operations.

4. **Detailed Parameter Definitions:** Complex expressions and calculations were broken down into simpler, more detailed parameter definitions. This granularity improved the transparency of code operations to ScaleHLS, aiding in more effective optimization.

After these transformations, the code becomes compatible with ScaleHLS. The following listing shows the ScaleHLS-compilable version of the `ped_subtract` function, demonstrating the applied modifications:

Listing 3.4: ScaleHLS-compatible version of the `ped_subtract` function

```
void ped_subtract(uint16_t results[5][NUM_SAMPLES][NUM_CHANNELS],
                  Uint16_t samples[NUM_SAMPLES * NUM_CHANNELS],
                  uint16_t all_peds[2][NUM_SAMPLES][NUM_CHANNELS],
                  uint8_t bank, unsigned starting_sample_number,
                  unsigned a) {
#pragma scop

    for (unsigned i = 0; i < NUM_SAMPLES; ++i) {
        unsigned idx = starting_sample_number;
        unsigned idx1 = idx % NUM_SAMPLES;
        if (idx >= NUM_SAMPLES){
            idx1 = idx - NUM_SAMPLES;
        }
        else{
            idx1 = idx;
        }
        for (unsigned c = 0; c < NUM_CHANNELS; ++c) {
            unsigned sample_idx = i * NUM_CHANNELS + c;
            unsigned first = samples[sample_idx];
            unsigned second = all_peds[bank][idx1][c];
            results[a][i][c] = second;
        }
    }
#pragma endscop
}
```

These code transformations went beyond mere syntactical changes, significantly impacting the hardware design process. Aligning the code with ScaleHLS's optimization paradigms resulted in hardware that was not only more efficient but also conformed better to the resource constraints of FPGA devices.

In conclusion, the code preparation for ScaleHLS involved a detailed review and transformation of the code's structural and operational aspects, ensuring compatibility and optimized

synthesis. This meticulous preparation facilitated the successful application of ScaleHLS in the project's subsequent stages.

3.2.3 Generation of Intermediate MLIR

The transformation of C code into an intermediate representation using MLIR (Multi-Level Intermediate Representation) is a crucial step in the ScaleHLS methodology. MLIR is an open-source project within the LLVM ecosystem designed to bridge the gap between high-level optimizations and low-level code generations across different hardware targets.

Following the modifications made for ScaleHLS compatibility, the next step is to generate the MLIR code, which represents a middle-ground form that captures the program's semantics in a way that is conducive to high-level optimizations and transformations.

To generate MLIR from the ScaleHLS-compatible C code, the following command is used in accordance with the instructions provided in the ScaleHLS repository:

Listing 3.5: ScaleHLS tool to Generate MLIR Code

```
$ cgeist ped_subtract.c -function=ped_subtract -S
    -memref-fullrank -raise-scf-to-affine > ped_subtract.mlir
```

In this command, `cgeist` is a tool that parses C/C++ code into MLIR, `ped_subtract.c` is the input file, and `ped_subtract.mlir` is the output file containing the MLIR code. The flags and options specify how the translation should be performed, focusing on the function to convert and the optimizations to apply.

This MLIR code serves as a pivotal element in the ScaleHLS process, laying the groundwork for further high-level synthesis optimizations and analysis. It abstracts away the intricacies of the specific hardware, allowing ScaleHLS to perform optimizations that are agnostic of the eventual target device, thus promoting greater efficiency and portability.

Here, it would be beneficial to include a snippet of the generated MLIR code to give the reader a concrete example of what MLIR looks like and how it represents the transformed C code.

Listing 3.6: Example of MLIR code generated by ScaleHLS for `ped_subtract`

```
module attributes {dlti.dl_spec =
    #dlti.dl_spec<#dlti.dl_entry<"dlti.endianness","little">,
    #dlti.dl_entry<i64, dense<64> : vector<2xi32>>,
    #dlti.dl_entry<f80, dense<128> : vector<2xi32>>,
    #dlti.dl_entry<i1, dense<8> : vector<2xi32>>,
    #dlti.dl_entry<i8, dense<8> : vector<2xi32>>,
    #dlti.dl_entry<i16, dense<16> : vector<2xi32>>,
    #dlti.dl_entry<i32, dense<32> : vector<2xi32>>,
    #dlti.dl_entry<f16, dense<16> : vector<2xi32>>,
    #dlti.dl_entry<f64, dense<64> : vector<2xi32>>,
    #dlti.dl_entry<f128, dense<128> : vector<2xi32>>>,
  llvm.data_layout =
    "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128",
  llvm.target_triple = "x86_64-unknown-linux-gnu",
    "polygeist.target-cpu" = "x86-64",
    "polygeist.target-features" = "+cx8,+fxsr,+mmx,+sse,+sse2,+x87",
    "polygeist.tune-cpu" = "generic"} {
  func.func @ped_subtract(%arg0: memref<5x256x16xi16>,
   %arg1: memref<256x16xi16>, %arg2: memref<4096xi16>,
   %arg3: i8, %arg4: i32, %arg5: i32)
   attributes {llvm.linkage = #llvm.linkage<external>} {
    %c-256_i32 = arith.constant -256 : i32
    %c256_i32 = arith.constant 256 : i32
    %0 = arith.index_cast %arg3 : i8 to index
    %1 = arith.index_cast %arg5 : i32 to index
    %2 = arith.cmpi uge, %arg4, %c256_i32 : i32
    %3 = scf.if %2 -> (i32) {
      %5 = arith.addi %arg4, %c-256_i32 : i32
      scf.yield %5 : i32
    } else {
      scf.yield %arg4 : i32
    }
    %4 = arith.index_cast %3 : i32 to index
    affine.for %arg6 = 0 to 256 {
      affine.for %arg7 = 0 to 16 {
        %5 = affine.load %arg1[%arg6, %arg7] : memref<256x16xi16>
        %6 = arith.extsi %5 : i16 to i32
        %7 = affine.load %arg2[%arg7 + symbol(%0) * 4096 + symbol(%4) * 16]
          : memref<4096xi16>
        %8 = arith.extsi %7 : i16 to i32
```

```
        %9 = arith.subi %6, %8 : i32
        %10 = arith.trunci %9 : i32 to i16
        affine.store %10, %arg0[symbol(%1), %arg6, %arg7]
          : memref<5x256x16xi16>
      }
    }
    return
  }
}
```

Understanding MLIR's role in the ScaleHLS ecosystem is crucial for comprehending the subsequent steps in the synthesis process, where these intermediate representations are further optimized and eventually translated into executable hardware configurations.

3.2.4 Optimized C++ Code Generation

After generating the MLIR, the next step in the ScaleHLS process is to transform this intermediate representation into optimized C++ code, particularly designed for HLS. This code is poised for synthesis into an FPGA bitstream. The transformation leverages the MLIR to apply various optimizations, culminating in C++ code that is efficient and tailored for HLS.

The command to generate optimized C++ code from the MLIR using ScaleHLS is:

Listing 3.7: ScaleHLS tool to Generate C++ Code

```
$ scalehls-opt ped_subtract.mlir -debug-only=scalehls
  -scalehls-dse-pipeline="top-func=ped_subtract␣target-spec=../config.json"
  | scalehls-translate -scalehls-emit-hlscpp > ped_subtract_dse.cpp
```

In this command:

- `scalehls-opt` processes the MLIR file (`ped_subtract.mlir`) to apply targeted optimizations.
- `-debug-only=scalehls` option enables debugging, offering insights into the optimization process.

- `-scalehls-dse-pipeline` specifies the design space exploration settings, with `top-func` indicating the primary function for optimization and `target-spec` referring to a configuration file that contains hardware-specific optimization parameters.
- `scalehls-translate` is used to convert the optimized MLIR into HLS-compatible C++ code, with `-scalehls-emit-hlscpp` ensuring the output is suitable for HLS tools.
- The final optimized C++ code is saved in `ped_subtract_dse.cpp`.

This critical stage bridges high-level programming constructs with hardware synthesis, yielding a refined C++ code that encapsulates the high-level optimizations from the MLIR. It exemplifies the capacity of ScaleHLS to automate and enhance the code optimization process, improving the performance and efficiency of the resultant hardware design.

To provide a tangible example, a snippet from the `ped_subtract_dse.cpp` file should be included in the thesis. This snippet will illustrate the type of transformations and optimizations performed by ScaleHLS, such as loop unrolling and array partitioning, which are essential for high-performance FPGA designs.

Listing 3.8: Optimized C++ code generated by ScaleHLS for `ped_subtract`

```
void ped_subtract(
  ap_int<16> v0[5][256][16],
  ap_int<16> v1[256][16],
  ap_int<16> v2[4096],
  ap_int<8> v3,
  ap_int<32> v4,
  ap_int<32> v5
) {     // L5, [0,38)
  #pragma HLS interface s_axilite port=return bundle=ctrl
  #pragma HLS interface s_axilite port=v3 bundle=ctrl
  #pragma HLS interface s_axilite port=v4 bundle=ctrl
  #pragma HLS interface s_axilite port=v5 bundle=ctrl
  #pragma HLS array_partition variable=v0 cyclic factor=16 dim=2
  #pragma HLS array_partition variable=v0 cyclic factor=8 dim=3
  #pragma HLS resource variable=v0 core=ram_t2p_bram

  #pragma HLS array_partition variable=v1 cyclic factor=16 dim=1
  #pragma HLS array_partition variable=v1 cyclic factor=8 dim=2
  #pragma HLS resource variable=v1 core=ram_t2p_bram

  #pragma HLS array_partition variable=v2 cyclic factor=8 dim=1
  #pragma HLS resource variable=v2 core=ram_t2p_bram

  int v6 = v3;  // L8, [0,0)
  int v7 = v5;  // L9, [0,0)
  bool v8 = v4 >= (ap_int<32>)256;      // L10, [0,0)
  ap_int<32> v9;
  if (v8) {     // L11, [0,0)
    ap_int<32> v10 = v4 + (ap_int<32>)-256;     // L12, [0,0)
    v9 = v10;   // L13, [0,0)
  } else {
    v9 = v4;    // L15, [0,0)
  }
  .....
```

Through this automated process, ScaleHLS significantly simplifies the generation of optimized C++ code, aligning with the synthesis requirements and enabling efficient FPGA implementation, thereby showcasing a vital step in the HLS workflow.

3.2.5 Synthesis with Vitis HLS

Vitis HLS, a high-level synthesis tool by AMD Xilinx, stands as a critical component in the hardware design workflow. It empowers developers to transform C/C++ code into FPGA hardware description language (HDL) effectively. This tool plays an indispensable role in enabling the synthesis of complex, algorithmic descriptions into tangible digital hardware configurations. It is particularly noted for its automation capabilities in optimizing and translating intricate algorithms into efficient hardware designs.

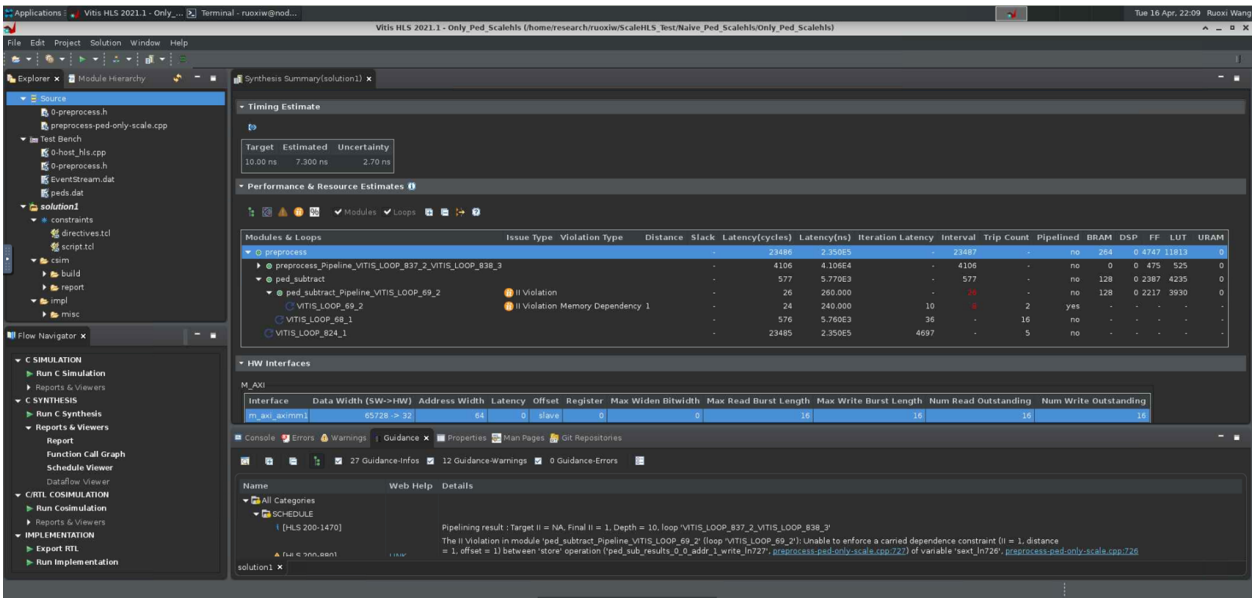

Figure 3.1: Vitis HLS Report Interface showing performance and resource estimates.

As illustrated in Figure 3.1, the Vitis HLS interface provides immediate insights into both the speed and resource utilization of the synthesized hardware design. It displays a comprehensive report, detailing the timing estimates, resource estimates, and the loop and task analysis, which are invaluable for performance tuning and optimization.

Within the ADAPT project, Vitis HLS is utilized for synthesizing the ScaleHLS-optimized C++ code into a HDL. This process is vital for assessing both the performance and the resource efficiency of the design when subject to the demanding conditions encountered in the field. Chapter 4 delves deeper into this, offering an in-depth discussion on performance evaluation and results, supplemented by detailed synthesis instructions and a complete simulation workflow.

Modifications to the code and testbench are integral for successful synthesis and simulation in Vitis HLS. These changes are geared towards ensuring seamless integration and evaluating the performance of a collection of preprocessing codes. The modifications made are as follows:

1. **Uniform Data Types:** The original ScaleHLS optimizations introduced data types like `ap_int`, which, while efficient for hardware, can create inconsistencies within a broader codebase that utilizes standard C++ types. To ensure uniformity across the system and facilitate integration, these have been shifted to `unsigned int`. This change aids in preventing parameter passing errors and provides a consistent platform for simulation.

2. **Streamlined Function Interface:** The `ped_subtract` function, a core component of the preprocessing, has been simplified. Earlier versions accepted more information than necessary, which could lead to inefficiencies. By streamlining the function to only process essential data, the design is made cleaner and execution more efficient.

These strategic changes fine-tune the code for the final synthesis step with Vitis HLS, analogous to setting the stage before the culminating performance. It ensures that the code not only integrates well but is also poised for optimal performance evaluation in the subsequent analysis phase.

For our ADAPT code, the sole modification involved changing the data type from `ap_int` to `unsigned` to ensure consistency across the full code.

Listing 3.9: Modified C++ `ped_subtract` Code

```
void ped_subtract(
  //ap_int<16> v0[NUM_ALPHAS][NUM_SAMPLES][NUM_CHANNELS], //v0[5][256][16],
  unsigned v0[NUM_ALPHAS][NUM_SAMPLES][NUM_CHANNELS],
  unsigned v1[256][16],
  unsigned v2[4096], //ap_int<16> v2[4096],
  unsigned v3,
  unsigned v4,
  unsigned v5
)
......
```

Additionally, we modified the main function to streamline parameter passing to the `ped_subtract` function. Instead of passing the entire structure, we now pass only the three parameters that are utilized, as detailed in the list below.

Listing 3.10: Modified main function Code

```
struct SW_Data_Packet *current_packet = &input_data_packet[alpha];
uint8_t bank = current_packet->bank;
uint8_t starting_sample_number = current_packet->starting_sample_number;
ap_int<16> temp_samples[256][16];
for (int i = 0; i < 256; i++) {
    for (int j = 0; j < 16; j++) {
    temp_samples[i][j] =
        static_cast<ap_int<16>>(current_packet->samples[i][j]);
    }
}
ped_subtract(ped_sub_results, temp_samples,
             input_all_peds, bank, starting_sample_number, alpha);
```

For a comprehensive understanding of how these modifications influence the synthesis output and performance metrics, refer to the detailed analysis in Chapter 4.1.

This section provides an overview of the synthesis process using Vitis HLS and highlights the necessary code modifications to facilitate accurate simulation and performance evaluation in the context of ScaleHLS optimizations.

3.3 Additional Code Transformations Examples

Following the methodology used in the `ped_subtract` algorithm, this section highlights the transformation process for the Signal Integration and Zero Suppression algorithms within the ADAPT project. Adapting these algorithms to comply with the ScaleHLS tool's requirements was crucial for achieving efficient FPGA synthesis. This section presents the ScaleHLS-compatible C code for the `integrate` and `zero_suppress` functions, which are prepared for input into the tool. The ScaleHLS tool then automatically generates the optimized final code.

3.3.1 Signal Integration

The Signal Integration algorithm, essential for boosting the signal-to-noise ratio, sums data points within a defined temporal window. This algorithm's transformation emphasized loop restructuring to exploit parallelism, alongside refined data management to improve memory access patterns on the FPGA. These modifications were critical for leveraging the parallel processing capabilities of FPGAs, thereby enhancing the algorithm's efficiency and performance in hardware.

Listing 3.11: ScaleHLS-compatible version of `integrate` Code

```
void integrate(int fine_time, int starting_sample, int bounds[8],
        int integrals[NUM_INTEGRALS * NUM_CHANNELS],
        uint16_t ped_sub_results[NUM_ALPHAS][NUM_SAMPLES][NUM_CHANNELS],
        unsigned a) {
#pragma scop
  int offset = fine_time - starting_sample;
  if (offset < 0) {
    offset += NUM_SAMPLES;
  }

  for (int s = 0; s < NUM_SAMPLES; ++s) {
    int x = s - offset;
    if (x < 0) {
      x += NUM_SAMPLES;
    }

    for (unsigned c = 0; c < NUM_CHANNELS; ++c) {
      for (unsigned i = 0; i < NUM_INTEGRALS; ++i) {
        int start = bounds[2*i];
        int end = bounds[2*i + 1];

        if ((x >= start && x <= end) ||
          ((x - NUM_SAMPLES) >= start)) {
          integrals[i * NUM_CHANNELS + c] += ped_sub_results[a][s][c];
        }
      }
    }
  }
#pragma endscop
}
```

3.3.2 Zero Suppression

Zero Suppression serves a pivotal role in data volume management by discarding trivial data points that do not exceed a predefined significance threshold. This is crucial for optimizing the storage and processing of on-chip data. Code transformations for this algorithm were targeted at enhancing conditional checks and streamlining the process of disregarding inconsequential data, reducing the FPGA's memory footprint.

Listing 3.12: ScaleHLS-compatible version of `zero_suppress` Code

```
void zero_suppress(int integrals[NUM_INTEGRALS * NUM_CHANNELS],
                   int thresholds[NUM_INTEGRALS]) {
#pragma scop
    for (unsigned i = 0; i < NUM_INTEGRALS; ++i) {
        for (unsigned c = 0; c < NUM_CHANNELS; ++c) {
            if (integrals[i * NUM_CHANNELS + c] < thresholds[i]) {
                integrals[i * NUM_CHANNELS + c] = 0;
            }
        }
    }
#pragma endscop
}
```

The enhancements to these algorithms showcase the adaptability of the ScaleHLS tool for diverse scenarios. By optimizing for ScaleHLS, the FPGA synthesis process is fine-tuned, achieving greater efficiency and better utilization of the FPGA's resources.

Chapter 4

Performance Evaluation and Results

4.1 Synthesis and Simulation with Vitis HLS

This section details the synthesis and simulation process using Vitis HLS, which is crucial for validating the performance and resource utilization of the hardware design generated from the ScaleHLS-optimized C++ code.

Setting Up the Vitis HLS Project: To assess the performance and efficacy of the synthesized design, a project is created in Vitis HLS. The setup involves the following steps:

1. Open Vitis HLS and create a new project.
2. Specify the project name and location.
3. Add the optimized C++ code (e.g., `ped_subtract_dse.cpp`) to the project.
4. Define the top function and add any necessary testbench files for simulation.
5. Set the clock period and select the FPGA target platform, such as the Kintex-7 KC705 Evaluation Platform.
6. Finalize the project setup to proceed with the synthesis and simulation.

Simulation and Synthesis Workflow: The project in Vitis HLS facilitates a comprehensive workflow to simulate and synthesize the design:

1. **C Simulation:** Run the C-level simulation to ensure the functional correctness of the design. This step compiles the C++ code and executes it to verify its behavior against the testbench.

2. **C Synthesis:** Perform synthesis to translate the C++ code into an FPGA-compatible design. This process analyzes the design's performance, including initiation intervals and resource utilization metrics such as BRAM, DSP, FF, and LUT.

3. **C/RTL Co-Simulation:** Conduct co-simulation to validate the synthesized design against a high-fidelity model of the FPGA hardware. This step provides detailed insights into the design's latency and operational characteristics.

The data acquired from these simulations are integral to understanding the design's performance and are used to compare against theoretical metrics and previous implementations. This comprehensive analysis ensures that the design not only meets the functional requirements but also adheres to the performance and resource constraints of the target FPGA platform.

In Chapter 4, the results of these simulations, alongside the detailed analysis of performance metrics and resource utilization, are discussed to provide a clear understanding of the design's efficacy and potential areas for further optimization.

4.2 Resource Utilization and Speed Analysis

This section presents a detailed analysis of resource usage and processing speed for FPGA architectures within our project, utilizing two distinct approaches to synthesis: the "Naive" approach and the "ScaleHLS" approach.

Naive Implementation: This term refers to the original code implementation without any optimization pragmas. It serves as our baseline, representing the raw, unoptimized performance and resource utilization of the FPGA when the code is synthesized exactly as initially written.

ScaleHLS Implementation: In contrast, the "ScaleHLS" approach involves using the ScaleHLS tool to compile the code. This method applies advanced synthesis techniques

that include automatic insertion of optimization directives, aiming to enhance both the computational efficiency and resource management of the FPGA.

The analysis begins by evaluating three key preprocessing functions—pedestal subtraction, signal integration, and zero suppression—individually. This initial assessment establishes a baseline for their standalone performance. Subsequently, we examine how these functions perform collectively when integrated, mirroring their operational use in the actual FPGA system. This comprehensive, dual-layered analysis offers insights into the overall efficiency and effectiveness of the combined code, highlighting the capabilities and limitations of our FPGA-based system in both "Naive" and "ScaleHLS" configurations.

4.2.1 Pedestal Subtraction

A comparative analysis of latency and resource utilization for the `ped_subtract` function is presented in Table 4.1, highlighting the native implementation's inefficiencies in FPGA architectures. These traditional methods do not fully leverage FPGA capabilities, leading to excessive latencies and poor resource allocation.

Conversely, applying ScaleHLS considerably improved this function's performance. Latency reduced from 4106 to 577 cycles, enhancing the process speed by approximately seven times. Despite ScaleHLS's increased resource demands compared to native methods, the trade-off is deemed beneficial, considering the substantial gains in performance while maintaining operational parameters.

Table 4.1: Resource and speed comparison of `ped_subtract`

Implementation	Latency	II	BRAM	DSP	FF	LUT
`Ped_Naive`	4 106	4 106	0	0	446	585
`Ped_ScaleHLS`	577	577	128	0	2 387	4 235

4.2.2 Integration

The `integrate` code analysis indicates an optimal solution, improving speed by nearly four-fold. Additionally, resource consumption decreased significantly, with reductions in Flip-Flops and Look-Up Tables by factors of 3.31 and 2.46, respectively, as detailed in Table 4.2.

Table 4.2: Resource and speed comparison of `integrate`

Implementation	Latency	II	BRAM	DSP	FF	LUT
`Integrate_Naive`	245 772	245 772	0	0	977	1 316
`Integrate_ScaleHLS`	65 545	65 545	0	0	295	535

4.2.3 Zero Suppression

These enhancements are not universally applicable. The `zero_suppress` function demonstrates the high-level synthesis complexity, with a 64 times speed reduction and higher resource utilization—Flip-Flops increased by 7.1 times and Look-Up Tables by 5 times. These findings emphasize the need for detailed analysis in each instance, highlighting the nuanced challenges of HLS tools as seen in Table 4.3.

Table 4.3: Resource and speed comparison of `zero_suppress`

Implementation	Latency	II	BRAM	DSP	FF	LUT
`Zero_Naive`	1 026	1 026	0	0	1 854	35 114
`Zero_ScaleHLS`	65 569	65 569	0	0	13 144	17 691

The bottom line is that we do not understand why this function performs so poorly relative to the others. The algorithm is not appreciably more complex (in fact, it is considerably simpler than `integrate`), yet the results are clearly not as good.

4.3 Comparison with Traditional HLS

The transition from traditional High-Level Synthesis (HLS) to ScaleHLS involves a nuanced balance of resource allocation and processing efficiency. Our comparative analysis revealed that while ScaleHLS demonstrated a 50% reduction in Flip-Flop and Look-Up Table utilization, it incurred a doubling in latency and a significant initiation interval expansion. This suggests that while ScaleHLS is more resource-efficient, it could introduce a trade-off in processing speed, especially evident with the inclusion of zero suppression algorithms.

To mitigate this, a 'tradeoff' approach was devised, combining the ScaleHLS-optimized code for pedestal subtraction and integration with a traditionally synthesized zero suppression code. This hybrid method aimed to harness the resource efficiency of ScaleHLS while offsetting the latency issues observed. The resulting performance showed an improvement in

speed and a reduction in resource use, yet the enhancement fell short of our expectations. This outcome indicates a complex interplay between data elements in the combined code, which necessitates a deeper dive to unravel the interactions and optimize the code further. This insight will steer the direction of future research and optimization efforts.

Table 4.4: FPGA synthesis metrics for full code implementations of different methodologies

Implementation	Latency	II	BRAM	DSP	FF	LUT
`Full_Traditional_HLS`	1 399 745	1 254 840	668	15	44 345	88 148
`Full_ScaleHLS`	2 036 368	10 182 084	292	15	25 684	42 426
`Full_Tradeoff`	1 971 825	9 859 369	292	15	14 392	28 300

4.4 Pros & Cons of ScaleHLS

ScaleHLS is compared with traditional HLS approaches to highlight its unique benefits and potential drawbacks.

Advantages of ScaleHLS ScaleHLS offers several advantages that enhance FPGA programming productivity and efficiency:

- **Automation of Code Conversion:** ScaleHLS excels in translating high-level C/C++ code into synthetically efficient hardware descriptions, reducing the time required for manual conversion and verification.
- **Resource Management:** It adeptly manages FPGA resources, making judicious use of the available gates and memory blocks, which is particularly beneficial for space-bound applications where physical resources are at a premium.
- **Optimization Techniques:** ScaleHLS applies advanced optimization techniques to improve the performance and efficiency of the synthesized hardware, potentially leading to faster computational speeds and lower power consumption.
- **Customizability:** Thanks to its modular architecture, ScaleHLS can be tailored to specific design requirements, offering flexibility to programmers in terms of both functionality and optimization.

Disadvantages of ScaleHLS Despite its advantages, ScaleHLS is not without its drawbacks:

- **Complexity of Auto-Generated Code:** The auto-generated code from ScaleHLS can be intricate and challenging to understand, making debugging and manual optimization more difficult for engineers.
- **Limited Applicability:** ScaleHLS may not always perform optimally across all types of codes and structures, requiring manual intervention or alternative methods for certain implementations.
- **Development Maturity:** Being a relatively new tool, ScaleHLS may lack the maturity and stability of established HLS tools, which might lead to unexpected behaviors or the need for frequent updates and patches.
- **Integration Challenges:** Integrating ScaleHLS-generated designs with existing systems and workflows can present challenges, particularly when dealing with legacy code or hardware constraints.

In conclusion, while ScaleHLS presents a forward-looking approach to high-level synthesis with its automated and optimization-centric design, it also necessitates a thoughtful application and consideration of its limitations. As the field of FPGA design evolves, tools like ScaleHLS will continue to shape the landscape of hardware synthesis, balancing automation with the artistry of engineering intuition and experience.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis has explored the integration of ScaleHLS into the ADAPT project, illustrating its impact on enhancing FPGA synthesis for space applications. The use of ScaleHLS has led to notable improvements in processing efficiency and resource management, particularly with the `ped_subtract` and `integrate` functions. By automating the translation of high-level C code into FPGA-optimized designs, ScaleHLS has effectively streamlined the synthesis process, accommodating the rigorous demands of space technology.

The findings underscore the substantial benefits that advanced high-level synthesis tools like ScaleHLS bring to FPGA programming. Not only do they enhance performance through sophisticated optimizations, but they also contribute to more resource-efficient designs, which are critical in resource-constrained environments like space.

Looking ahead, the insights gained from this study encourage further exploration and development of high-level synthesis tools. The potential for ScaleHLS to evolve and further refine its capabilities suggests a promising direction for future research. As these tools mature, they will undoubtedly broaden the scope of possibilities for FPGA applications, especially in critical sectors where precision and reliability are paramount. This progression is expected to foster significant advancements in space technology and beyond, pushing the boundaries of what can be achieved with FPGA-based systems.

5.2 Future Work

In future work, the focus will be on addressing performance constraints evident in current FPGA synthesis, as demonstrated by our results. A significant trade-off between speed and resource utilization persists, raising critical concerns. To address these challenges, the introduction of HIDA [19] (Hierarchical Dataflow Compiler for High-Level Synthesis) marks a significant advancement. Developed by the same team as ScaleHLS, HIDA leverages a new scalable and hierarchical HLS framework designed to systematically transform algorithmic descriptions into optimized dataflow implementations on hardware.

HIDA introduces an innovative dataflow Intermediate Representation (IR), known as HIDA-IR, which models dataflow at two distinct levels of abstraction—Functional and Structural—facilitating effective optimization strategies. These include a pattern-driven task fusion algorithm and an intensity- and connection-aware dataflow parallelization algorithm, which are designed to maximize efficiency. Additionally, HIDA supports an end-to-end and extensible compilation stack that accommodates inputs from both PyTorch and C++, empowering users to rapidly experiment with various design parameters and prototype new dataflow architectures. Comprehensive evaluations on FPGA platforms reveal that HIDA can achieve up to 8.54 times higher throughput compared to existing state-of-the-art HLS tools.

For the APT project, while ScaleHLS may not provide the optimal solution, HIDA's capabilities offer profound insights and guide further improvements. This includes exploring additional HLS tools to expand our range of possible solutions, thereby enhancing the overall design and effectiveness of FPGA implementations.

References

[1] James Buckley et al. The Advanced Particle-astrophysics Telescope (APT) Project Status. In *Proc. of 37th Int'l Cosmic Ray Conference*, volume 395, pages 655:1–655:9. Sissa Medialab, July 2021.

[2] Wenlei Chen et al. The Advanced Particle-astrophysics Telescope: Simulation of the Instrument Performance for Gamma-Ray Detection. In *Proc. of 37th Int'l Cosmic Ray Conference*, volume 395, pages 590:1–590:9. Sissa Medialab, 2021.

[3] Clayton J Faber, Steven D Harris, Zhili Xiao, Roger D Chamberlain, and Anthony M Cabrera. Challenges designing for FPGAs using high-level synthesis. In *Proc. of High Performance Extreme Computing Conference*. IEEE, September 2022.

[4] Daniel D Gajski, Nikil D Dutt, Allen CH Wu, and Steve YL Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[5] Maya Gokhale, Jan Stone, Jeff Arnold, and Mirek Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *Proc. of Symposium on Field-programmable Custom Computing Machines*, pages 49–56. IEEE, 2000.

[6] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *Proc. of IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[7] Marco Lattuada, Fabrizio Ferrandi, and Maxime Perrotin. Data transfers analysis in computer assisted design flow of FPGA accelerators for aerospace systems. *IEEE Transactions on Multi-Scale Computing Systems*, 4(1):3–16, 2017.

[8] Vasileios Leon, Ioannis Stamoulias, George Lentaris, Dimitrios Soudris, David Gonzalez-Arjona, Ruben Domingo, David Merodio Codinachs, and Isabelle Conway. Development and testing on the European space-grade BRAVE FPGAs: Evaluation of NG-large using high-performance DSP benchmarks. *IEEE Access*, 9:131877–131892, 2021.

[9] Konstantinos Maragos, Vasileios Leon, George Lentaris, Dimitrios Soudris, David Gonzalez-Arjona, Ruben Domingo, Antonio Pastor, David Merodio Codinachs, and Isabelle Conway. Evaluation methodology and reconfiguration tests on the new European NG-MEDIUM FPGA. In *Proc. of NASA/ESA Conference on Adaptive Hardware and Systems*, pages 127–134. IEEE, 2018.

[10] Vivek V Menon, Saquib A Siddiqui, Sanil Rao, Andrew Schmidt, Matthew French, Ved Chirayath, and Alan Li. Design and performance evaluation of multispectral sensing algorithms on CPU, GPU, and FPGA. In *Proc. of IEEE Aerospace Conference*. IEEE, 2021.

[11] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. Predictable accelerator design with time-sensitive affine types. In *Proc. of 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 393–407. ACM, 2020.

[12] Sebastian Sabogal and Alan George. A methodology for evaluating and analyzing FPGA-accelerated, deep-learning applications for onboard space processing. In *Proc. of IEEE Space Computing Conference*, pages 143–154. IEEE, 2021.

[13] Ahmed Sanaullah, Rushi Patel, and Martin Herbordt. An empirically guided optimization framework for FPGA OpenCL. In *Proc. of International Conference on Field-Programmable Technology*, pages 46–53. IEEE, 2018.

[14] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. AutoDSE: Enabling software programmers to design efficient FPGA accelerators. *ACM Transactions on Design Automation of Electronic Systems*, 27(4):32:1–32:27, 2022.

[15] Marion Sudvarg et al. Front-End Computational Modeling and Design for the Antarctic Demonstrator for the Advanced Particle-astrophysics Telescope. In *Proc. of 38th International Cosmic Ray Conference*, volume 444, pages 764:1–764:9. Sissa Medialab, July 2023.

[16] Marion Sudvarg, Chenfeng Zhao, Ye Htet, Meagan Konst, Thomas Lang, Nick Song, Roger D. Chamberlain, Jeremy Buhler, and James H. Buckley. HLS taking flight: Toward using high-level synthesis techniques in a space-borne instrument. In *Proc. of 21st International Conference on Computing Frontiers*. ACM, May 2024.

[17] Louis van Harten, Roel Jordans, and Hamid Pourshaghaghi. Necessity of fault tolerance techniques in Xilinx Kintex 7 FPGA devices for space missions: A case study. In *Proc. of Euromicro Conference on Digital System Design*, pages 299–306. IEEE, 2017.

[18] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, and D. Chen. ScaleHLS: A new scalable high-level synthesis framework on multi-level intermediate representation. In *Proc. of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 741–755, April 2022.

[19] Hanchen Ye, Hyegang Jun, and Deming Chen. HIDA: A hierarchical dataflow compiler for high-level synthesis. In *Proc. of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[20] Hanchen Ye, HyeGang Jun, Hyunmin Jeong, Stephen Neuendorffer, and Deming Chen. ScaleHLS: A scalable high-level synthesis framework with multi-level transformations and optimizations. In *Proc. of 59th ACM/IEEE Design Automation Conference*, pages 1355–1358. ACM, July 2022.

[21] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. AutoPilot: A platform-based ESL synthesis system. In Philippe Coussy and Adam Morawiec, editors, *High-Level Synthesis: From Algorithm to Digital Circuit*, pages 99–112. Springer, 2008.