WASHINGTON UNIVERSITY IN ST. LOUIS

McKelvey School of Engineering
Department of Computer Science and Engineering

Thesis Examination Committee:
Roger Chamberlain, Chair
James Buckley
Jeremy Buhler

Applying HLS to FPGA Data Preprocessing in the Advanced Particle-astrophysics Telescope

by

Meagan Sue Konst

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

December 2022
St. Louis, Missouri

Table of Contents

List of Figures iv

List of Tables vi

List of Abbreviations viii

Acknowledgments ix

Abstract xi

Chapter 1: Introduction 1
- 1.1 Computational Context and Requirements 3
- 1.2 Contributions and Thesis Outline 5

Chapter 2: Background and Related Work 7
- 2.1 Background 7
- 2.2 Related Work 9

Chapter 3: Accelerated Algorithms 11
- 3.1 Software Interface Functions 11
  - 3.1.1 data_packet_dat_to_struct 11
  - 3.1.2 peds_dat_to_arrays 12
  - 3.1.3 write_output 15
  - 3.1.4 struct_to_json 15
- 3.2 Hardware Functions 16
  - 3.2.1 ped_subtract 16
  - 3.2.2 integral 18

Chapter 4: HLS Implementation 20
- 4.1 Naive Approach 21
- 4.2 ped_subtract Optimizations 25
- 4.3 integral Optimizations 27

Chapter 5: Conclusions and Future Work 32
- 5.1 Conclusions 32

5.1.1 Comparison of Naive and Current Approach 32
5.1.2 Learnings 33
5.2 Future Work 33

References 36

List of Figures

Figure 1.1: Top: APT in Falcon-9 faring. Bottom: Detection modes. [2] 2

Figure 1.2: APT hardware pipeline. 3

Figure 1.3: The software model used to generate input data for the FPGA. . . . 4

Figure 1.4: The packet format used to communicate a single sampling window to the FPGA. 4

Figure 1.5: A sampling window showing the four different integral types: 1) blue: pre-signal noise, 2) purple: main signal, 3) green: tail, 4) red: whole window. 5

Figure 2.1: Characterizations of an imperfect loop. 8

Figure 2.2: Characterizations of a semi-perfect loop. 8

Figure 2.3: Characterizations of a perfect loop. 9

Figure 3.1: Software interface functions data flow diagram. 12

Figure 3.2: Part of the EventStream.dat input file. 13

Figure 3.3: SW_Data_Packet C struct definition. 13

Figure 3.4: Part of the peds.dat input file. 14

Figure 3.5: The data structure of the peds.dat file. 14

Figure 3.6: The 3D array representation of the pedestal values. 14

Figure 3.7: An example output.txt file with fabricated integral bounds. 15

Figure 3.8: A snippet of the packet.json file that is used for debugging and data dumps. 16

Figure 3.9: Visual representation of the ped_subtract function. 17

Figure 3.10: Software ped_subtract pseudocode. 17

Figure 3.11: Visual representation of the two integral function configurations: linear and wraparound. 18

Figure 3.12: Software integral pseudocode. 19

Figure 4.1: Connections between the Vitis Host and Kernel components. 22

Figure 4.2: Naive ped_subtract source code. 23

Figure 4.3: Naive integral source code. 24

Figure 4.4: The footprint of the U280 with the naive approach. The dark blue highlighted section represents the footprint of the Kernel. 25

Figure 4.5: Semi-perfect ped_subtract source code. 27

Figure 4.6: Ternary operators version of integral source code. 29

List of Tables

Table 4.1: Naive hardware emulation area statistics. 23

Table 4.2: Naive hardware emulation latency statistics. 26

Table 4.3: Naive hardware run area statistics. 26

Table 4.4: Naive hardware run latency statistics. 26

Table 4.5: ped_subtract with semi-perfect loop area statistics. 26

Table 4.6: ped_subtract with semi-perfect and perfect loop latency statistics. . . . 27

Table 4.7: ped_subtract with perfect loop area statistics. 28

Table 4.8: integral with forced perfect loop area statistics. 28

Table 4.9: integral with forced perfect loop latency statistics. 28

Table 4.10: integral with combined logic area statistics. 28

Table 4.11: integral with ternary operators area statistics. 30

Table 4.12: integral with ternary operators latency statistics. 30

Table 4.13: integral with incorrect dependency resolution area statistics. 30

Table 4.14: integral with incorrect dependency resolution latency statistics. . . . 30

Table 4.15: Current approach hardware emulation area statistics. 30

Table 4.16: Current approach hardware emulation latency statistics. 30

Table 4.17: Current approach hardware run area statistics. 31

Table 4.18: Current approach hardware run latency statistics. 31

Table 4.19: Current approach hardware run ped_subtract area statistics. 31

Table 4.20: Current approach hardware run integral area statistics. 31

List of Abbreviations

- ADAPT: Antarctic Demonstrator for the Advanced Particle-astrophysics Telescope
- APT: Advanced Particle-astrophysics Telescope
- AXI: Advanced eXtensible Interface
- BRAM: Block Random Access Memory
- DCGP: Deep Correlated Gaussian Process
- DSP: Digital Signal Processor
- FF: Flip-Flop
- FPGA: Field-Programmable Gate Array
- HBM: High-Bandwidth Memory
- HDL: Hardware Description Language
- HLS: High-Level Synthesis
- I/O: Input/Output
- JSON: JavaScript Object Notation
- MWI: Multiple Work Item
- SWI: Single Work Item
- URAM: Unified Random Access Memory
- VHDL: VHSIC Hardware Description Language
- VHSIC: Very High Speed Integrated Circuit
- WAR: Write After Read

Acknowledgments

I would first and foremost like to thank my family for their moral and financial support throughout my life and academic career. I would also like to thank all of my professors and teaching assistants for dedicating their time to provide me with a stellar engineering education. And of course, I would like to thank my academic advisors: Chris Ramsay, Matthew Lew, and Roger Chamberlain for all of their advice and council throughout my time at Washington University in Saint Louis. I would also like to acknowledge and thank the PhD students who have helped me with my thesis: Chenfeng Zhao, Marion Sudvarg, and Clayton Faber.

Meagan Sue Konst

Washington University in St. Louis
December 2022

Dedicated to my family and friends for their constant support.

ABSTRACT OF THE THESIS

Applying HLS to FPGA Data Preprocessing in the
Advanced Particle-astrophysics Telescope
by
Meagan Sue Konst
Master of Science in Computer Engineering
Washington University in St. Louis, 2022
Professor Roger Chamberlain, Chair

The Advanced Particle-astrophysics Telescope (APT) and its preliminary iteration the Antarctic Demonstrator for APT (ADAPT) are highly collaborative projects that seek to capture gamma-ray emissions. Along with dark matter and ultra-heavy cosmic ray nuclei measurements, APT will provide sub-degree localization and polarization measurements for gamma-ray transients. This will allow for devices on Earth to point to the direction from which the gamma-ray transients originated in order to collect additional data. The data collection process is as follows. A scintillation occurs and is detected by the wavelength-shifting fibers. This signal is then read by an ASIC and stored in an ADC buffer. This buffer is then formatted as a data packet with a meaningful header and stored in memory that is accessible by an FPGA. This is where the data must be preprocessed before being sent on to the CPU for the localization algorithms. This preprocessing includes capacitive memory pedestal subtraction and taking four trigger-relative time integrals per channel. There are 16 channels per ASIC. The HLS implementation of this FPGA data preprocessing seeks to answer the following questions. How well can HLS map this naïve C preprocessing model to an FPGA image? What HLS optimizations are most useful for this application? What are the reported latencies of these optimized models? How much chip area is consumed by each of these designs? How many ASICs can be processed by one FPGA?

Chapter 1

Introduction

The Advanced Particle-astrophysics Telescope (APT) and its preliminary iteration the Antarctic Demonstrator for APT (ADAPT) are highly collaborative projects that seek to capture gamma-ray emissions. Along with dark matter and ultra-heavy cosmic ray nuclei measurements, APT will provide sub-degree localization and polarization measurements for gamma-ray transients. This will allow for devices on Earth to point to the direction from which the gamma-ray transients originated in order to collect additional data. A version of the flight instrument and its three detection modes is shown in Figure 1.1 [2]. The focus for this thesis is the preprocessing for the gamma-ray detection and localization computational pipeline. Descriptions of the overall computational pipeline are provided in [15, 16]. This thesis has its focus on the processing prior to this, essentially the data preprocessing steps prior to the centroiding stage.

Here, we will explore the development of the data preprocessing using High-Level Synthesis (HLS), which holds the promise of decreasing development effort for FPGA designs [5, 12], but is fraught with issues of non-performing applications [13]. The HLS implementation of this FPGA data preprocessing seeks to answer the following questions.

- How well can HLS map this naïve C preprocessing model to an FPGA image?
- What HLS optimizations are most useful for this application?
- What are the reported latencies of these optimized models?
- How much chip area is consumed by each of these designs?
- How many ASICs can be processed by one FPGA?

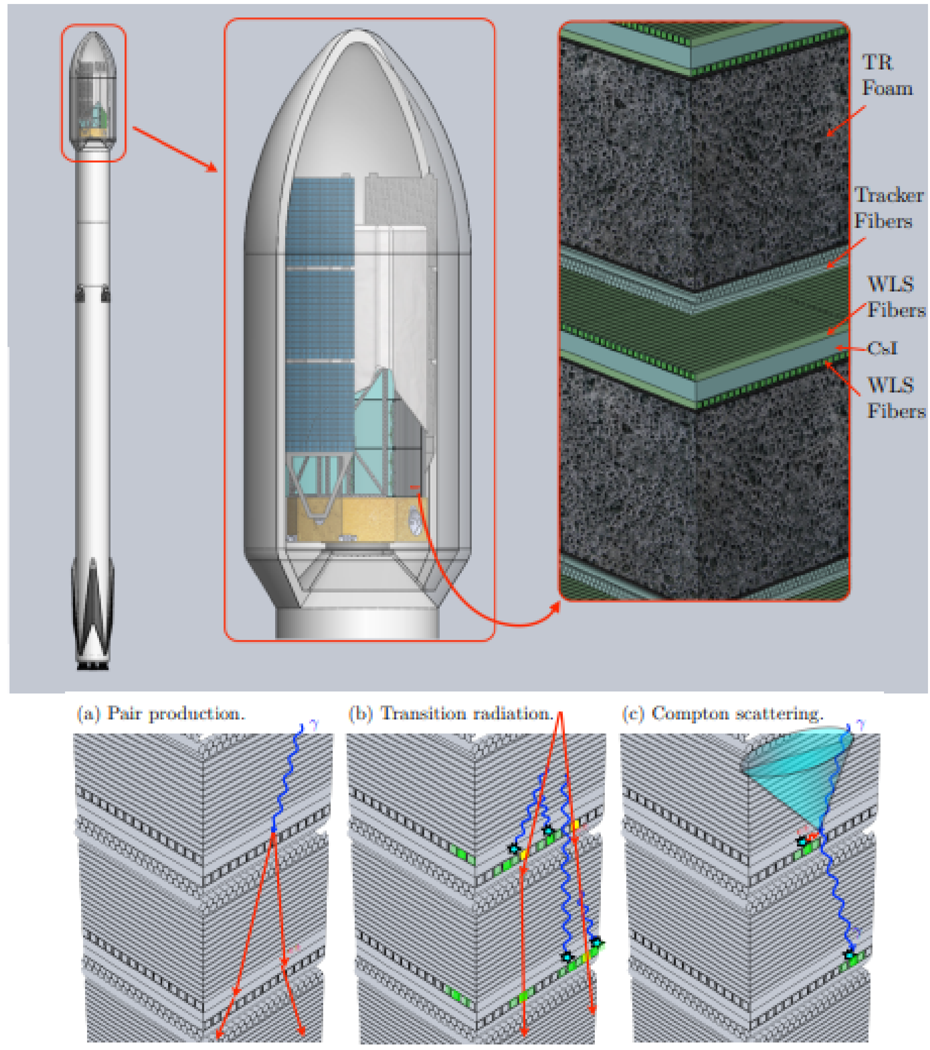

Figure 1.1: Top: APT in Falcon-9 faring. Bottom: Detection modes. [2]

1.1 Computational Context and Requirements

The APT data collection process is as follows. A scintillation occurs and is detected by the wavelength-shifting fibers. This signal is then read by an ALPHA ASIC and stored in an ADC buffer. This buffer is then formatted as a data packet with a meaningful header and stored in memory that is accessible by an FPGA. This is where the data must be preprocessed before being sent on to the CPU for the localization algorithms. The APT hardware pipeline used for the data collection described above is shown in Figure 1.2.

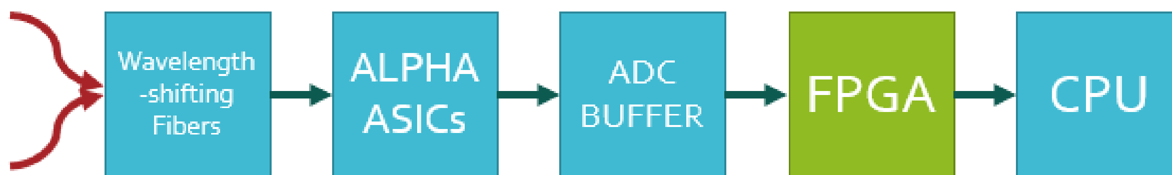

Figure 1.2: APT hardware pipeline.

Within the APT software pipeline that is used for testing purposes, there is a set of C functions that generates input data for the FPGA. These generation files were originally developed by Gary Varner and his team at The University of Hawai'i and have since been modified by Marion Sudvarg from Washington University in St. Louis to include the most recent expected input signal models developed by Leonardo Di Venere and their team at National Institute for Nuclear Physics - Bari [1].

The full input generation system is shown in Figure 1.3 with the inputs to the FPGA (peds.dat and EventStream.dat) circled and the pre-existing FPGA emulation crossed out since it is being replaced by the work described in this thesis.

The EventStream.dat file is a space-delineated bit stream that requires the corresponding packet definition shown in Figure 1.4 in order to be properly interpreted. We have highlighted the fields within this packet definition that are most relevant to our work:

- Bank ID: Specifies which ADC buffer bank (A or B) the packet originated from. Having two banks allows for one bank to be filled while the other is being read out.
- FineTime: The sample number when the trigger arrives. The trigger occurs once the event is detected.

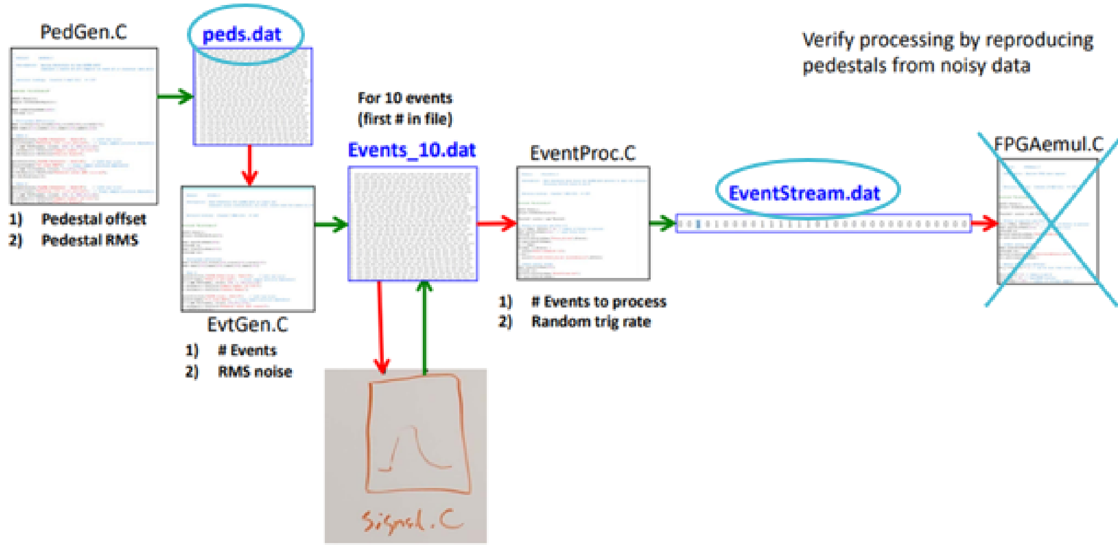

Figure 1.3: The software model used to generate input data for the FPGA.

- N samples to be read: How many samples within the ADC buffer were valid / relevant.
- Starting sample number: The sample number when the first component of the event arrives.
- Samples: The samples for each of the 16 channels collected every 10 ns.

Word #	Category	16-bit Field	Bit-wise Field	Comment
1	Start Word	0xA1FA		ALphA
2	ASIC ID/FineTime		xxxZZZZB ssssssss	x=I2C ADDR, Z=Conf Addr (8x), B=Bank A/B, s=FineTime (sample number when trigger arrives)
3	Coarse Time 2	msw	Mccc cccc cccc cccc	32 bit coarse time counter, increments every time 256 sample pointer wraps around
4	Coarse Time 1	lsw	cccc cccc cccc cccL	
5	Trigger Number		Mnnn nnnn nnnn nnnL	16 bit count of the number of received triggers
6	Acquistion Window		Mttt tttL Mlll lllL	t=SamplesAfterTrigger, l=LookBackSamples; M=Msb, L=Lsb
7	Samples to Rd/Start		Mrrr rrrL Mppp pppL	r= N samples to be read, p=Starting sample number
8	Status word		Mxxx xxxL Mmmm mmmL	x= Number of missed triggers, m= State machine status
9	Smp p, Ch 0		hhhh Maaa aaaa aaaL	h = channel, a = ADC value
10	Smp p, Ch 1		0001 aaaa aaaa aaaa	
11	Smp p, Ch 2		0010 aaaa aaaa aaaa	
12	Smp p, Ch 3		0011 aaaa aaaa aaaa	
...				
23	Smp p, Ch 14		1110 aaaa aaaa aaaa	
24	Smp p, Ch 15		1111 aaaa aaaa aaaa	
25	Smp p+1, Ch 0		0000 aaaa aaaa aaaa	
26	Smp p+1, Ch 1		0001 aaaa aaaa aaaa	
...				
8+N*16-1	Smp p+N, Ch 14		1110 aaaa aaaa aaaa	
8+N*16	Smp p+N, Ch 15		1111 aaaa aaaa aaaa	
8+N*16+1	Stop Word	0x0E6A	OmEGA	

Figure 1.4: The packet format used to communicate a single sampling window to the FPGA.

The main components of the current data preprocessing scheme that is being implemented on the FPGA are as follows: pedestal subtraction and performing four trigger-relative time

integrals for each channel. There are 16 channels per ALPHA ASIC. A pedestal is an intentional offset that is added to our signal in order to be able to detect signals of very small magnitudes. This offset is unique and known for each of the ADC buffer slots. Pedestal subtraction removes these offsets after they have served their purpose in generating a signal large enough to create a trigger. An integral over time provides us with the total energy produced by an event over a given time period. There are currently four different integral types of interest: pre-signal noise, main signal, tail of the signal, and whole sampling window. These four integral types are shown on a fake input signal in Figure 1.5.

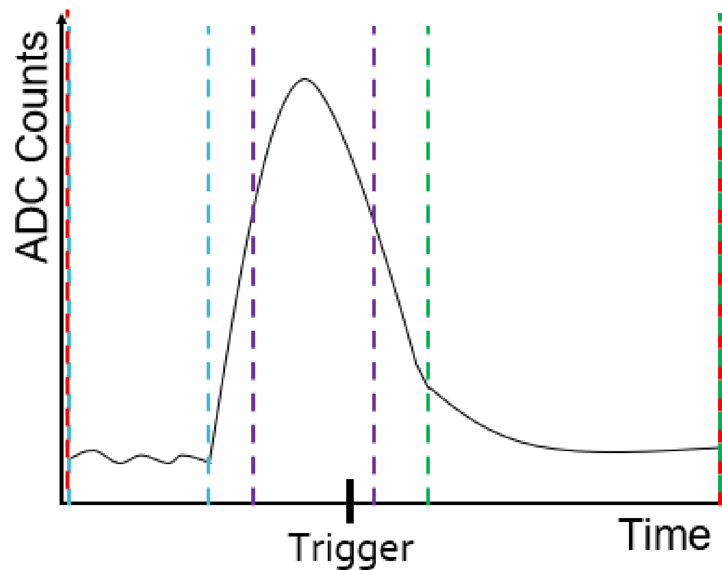

Figure 1.5: A sampling window showing the four different integral types: 1) blue: pre-signal noise, 2) purple: main signal, 3) green: tail, 4) red: whole window.

1.2 Contributions and Thesis Outline

This thesis makes the following contributions to the ADAPT mission:

- A software testing infrastructure to interface with the existing input generation system and the new data preprocessing functions.
- Software implementations of the data preprocessing functions that are used to verify functional correctness.

- Vitis Host and Kernel files to map the software implementation to an FPGA image.
- Reports on how well HLS can map the naive C preprocessing model to an FPGA image.
- Clocking frequency, latency, area, and power measurements of several HLS optimization iterations.
- Estimates for how many ASICs an FPGA can support based on area statistics.
- A hardware implementation of the data preprocessing functions that is ready for further HLS optimizations.
- Suggestions on which HLS optimizations to try next.

The outline of the remainder of the thesis is as follows. Chapter 2 provides background and related work. Chapter 3 describes the algorithms that are accelerated on the FPGA. Chapter 4 gives the HLS implementations, and Chapter 5 has conclusions and a brief description of future work.

Chapter 2

Background and Related Work

2.1 Background

The first concept that needs to be discussed is the two main FPGA design approaches. The traditional approach for designing an FPGA image involves using hardware description languages (HDLs) such as Verilog or VHDL. The main issues with this approach are that there is a steep learning curve for using these HDLs, this approach is prone to user error, and it is also very difficult to debug [8]. The main way to debug HDL implementations is waveform analysis, where every relevant signal is examined relative to the system clock(s). The second, newly available approach to FPGA design is High-Level Synthesis (HLS). This approach allows software engineers to produce FPGA images using software languages such as C, C++, and OpenCL C.

There are two major FPGA suppliers, each with their own set of HLS tools: Xilinx and Intel. We are focusing on Xilinx devices and are thus using their Vitis toolset for HLS. Vitis HLS implementations are divided into two components: the Host and the Kernel. The Host is responsible for creating an OpenCL environment, initializing the inputs to the kernel as memory buffers, initiating the Kernel, and reading and processing its outputs. This is where the software interface functions described in Chapter 3 are implemented. The Kernel is the equivalent of a FPGA module where the preprocessing algorithms (pedestal subtraction and integration) are implemented. The Host and Kernel communicate over a specified interface. The default interface is AXI, which is a standard ARM bus-protocol.

Another important topic to discuss for HLS optimization is loop type. There are three types of loops: imperfect, semi-perfect, and perfect. For HLS optimizations, a semi-perfect or perfect loop is preferred [18]. An imperfect loop is characterized by having variable loop bounds and operations outside of the innermost loop, as shown in Figure 2.1. A semi-perfect

loop has all of its operations inside the innermost loop and has variable bounds only on its outermost loop, as shown in Figure 2.2. A perfect loop has all of its operations inside the innermost loop and has constant bounds for all loops, as shown in Figure 2.3. It is important to note that perfect loops may require conditional execution within the innermost loop if the given operation should not be performed every iteration. Semi-perfect and perfect loops have similar performance metrics and so whichever type is more applicable for the use-case should be used.

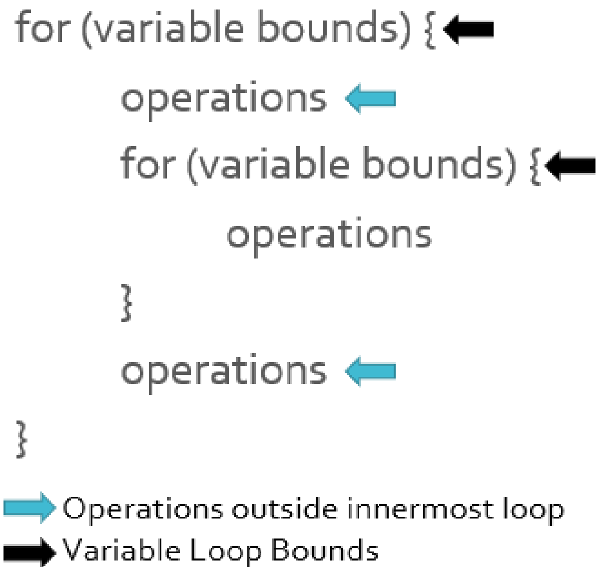

Figure 2.1: Characterizations of an imperfect loop.

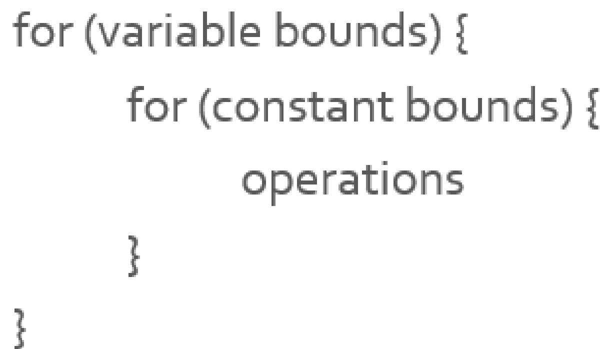

Figure 2.2: Characterizations of a semi-perfect loop.

```
for (constant bounds) {
    for (constant bounds) {
        operations
    }
}
```

Figure 2.3: Characterizations of a perfect loop.

2.2 Related Work

While the tools and methods to use HLS are still being developed, there has been a study that compares the design productivity of HLS compared to HDL where they report an average productivity gain of 2.3x, which breaks down into an average gain in design time of 4.4x and an average loss in quality of 1.9x [12].

A common theme across the HLS research space is that direct porting of a design from a different platform is nowhere near enough to achieve similar performance [4, 11, 13, 20, 21]. Several optimizations must be applied and even then, the HLS solution may perform worse than the mature software solution or exhibit variable performance depending on the input size while the software solution has uniform performance across all inputs [11].

There are two main optimization styles for HLS: Multiple Work Item (MWI) and Single Work Item (SWI). MWI refers to acting on wide vectorized data whereas SWI refers to creating a deeply pipelined solution. While there has been work investigating MWI versus SWI styles [3, 11], this thesis stays with the SWI style as recommended by Xilinx [18].

In order to familiarize ourselves with the current HLS optimization techniques, we watched a lecture given by Johannes de Fine Licht covering the material from their survey paper. In the lecture, several concepts and techniques were discussed, such as the data flow pipeline, initiation intervals, the depth and width of an operation, loop unrolling, and potential fanout issues that can limit resource utilization [6].

The major barrier for using HLS is the knowledge and skill of proper pragma placement. To this end, there has been a fair amount of work attempting to set pragmas automatically [14, 17]. One of these papers has taken a bottleneck-centric optimization approach and has been able to provide performance-comparable solutions with a 26.38x pragma reduction compared to manually optimized HLS solutions [14]. The other has has chosen a Deep Correlated Gaussian Process (DCGP) optimization model [17]. Along with pragma placement automation, there has also been work towards automated debugging based on the functional equivalence of software and hardware implementations that allows for error localization and correction [8].

While there have been several papers documenting best practices when it comes to HLS, there is still an ongoing issue of being able to predict performance at compile time [7]. Some of these failed optimization attempts have been documented by our group to provide areas for improvement for the toolchain development, along with a recommendation for the continuation of making parts of the HLS toolchain open-source [7].

Chapter 3

Accelerated Algorithms

Two sets of algorithms are necessary to interact with the Vitis infrastructure. The first set is implemented in the Host component and contains the software interface functions that are needed to connect the hardware functions to the larger software pipeline model. The second set is implemented in the Kernel component and contains the hardware functions that perform the main data preprocessing. The Host functions are solely used as test infrastructure, while the Kernel functions will actually be deployed on the flight instrument. The following sections dive into more detail about both sets of algorithms.

3.1 Software Interface Functions

The software interface functions serve to connect the hardware model of the data preprocessing functions to the larger software pipeline model for event localization. These functions are for testing purposes only and will not be deployed on the flight instrument. The data flow of inputs and outputs for the software interface functions are shown in Figure 3.1. The generation of the EventStream.dat and peds.dat files is discussed in Chapter 1 and shown in Figure 1.3.

3.1.1 data_packet_dat_to_struct

The first function that will be discussed is data_packet_dat_to_struct. As the name would imply and as is shown in Figure 3.1, this function takes the EventStream.dat file as an input and produces a usable struct representation of that data. The EventStream.dat file is a space-delineated bit file, which is difficult to interpret at a glance, as shown in Figure 3.2.

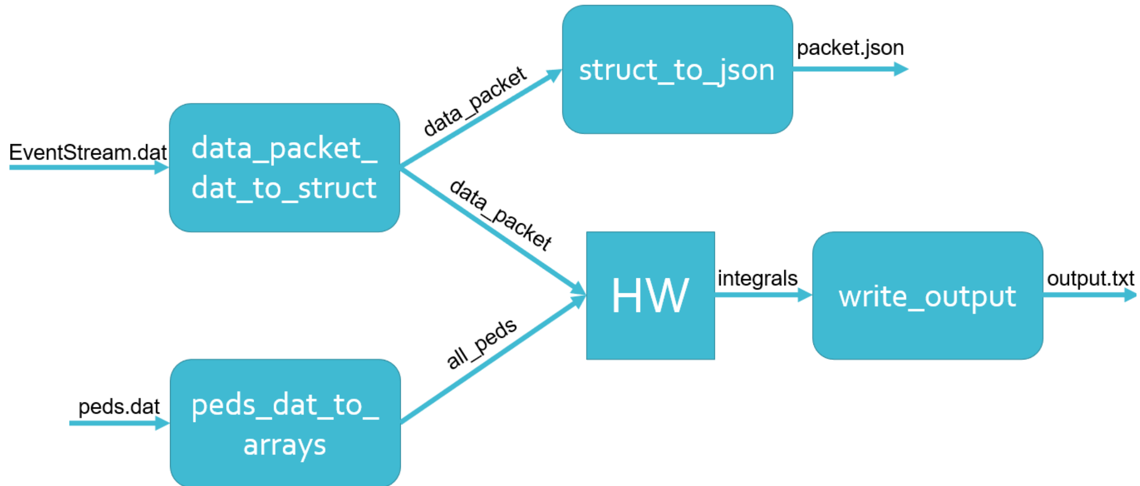

Figure 3.1: Software interface functions data flow diagram.

The resulting output of this function is much easier to work with and interprets the bit file as the data packet described in Figure 1.4 but with each component mapped to its own field in a C struct type. The struct definition used is shown in Figure 3.3.

3.1.2 peds_dat_to_arrays

Similar to data_packet_dat_to_struct, peds_dat_to_arrays takes a .dat file as an input and provides a more useful C configuration as an output. In this case, that output is a 3D array containing all of the pedestal values for both ADC buffer banks. The input file peds.dat is more readable than the EventStream.dat file due to its configuration. Each row in the file starts with a piece of metadata that specifies the sample number. This metadata is then followed by the 16 channel values for that sample. Along with this helpful metadata, all values are reported in decimal format as opposed to bit format, which again helps with readability. A snippet of the peds.dat file is shown in Figure 3.4. The full data structure of the file is shown in Figure 3.5 where A and B represent the two ADC buffer banks. From this structure it is simple to see the appropriate mapping to the 3D array shown in Figure 3.6.

```
1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 1 1 1 0 1 0 1 0 1 0 0 0 0 0 0 1 1 1 1 0 1 1 0 0 1
0 0 0 0 0 0 1 1 1 1 1 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 1 1
1 1 1 0 0 1 0 0 0 0 0 0 0 0 1 1 1 1 1 0 1 1 0 1 0 0 0 0 0 0 1 1 1 1 1 1 0 1 1 0
0 0 0 0 0 0 1 1 1 1 1 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 1 0 0 1 1 1 0 0 0 0 0 0 1 1
1 1 1 0 1 0 0 1 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 1 1 1 1 0 1 0 1 0 1
0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 1 0 0 0 0 0 0 1 1 1 1 0 1 1 1 1 1 0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 1 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1
0 0 0 0 0 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 1 1 0 1 0 0 0 0 0 0 1 1
1 1 1 0 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 1 0 0 0 0 0 0 1 1 1 0 1 1 1 0 1 1
0 0 0 0 0 0 1 1 1 1 1 0 0 1 0 1 0 0 0 0 0 0 1 1 1 1 0 1 0 1 1 1 0 0 0 0 0 0 1 1
1 1 1 0 1 1 1 0 0 0 0 0 0 0 1 1 1 1 0 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 0 1 0 0 0
0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 1 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1
1 1 1 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 1 0 0 0 0 0 0 1 1 1 1 0 1 1 1 1 0
0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 0 1 0 0 0 0 0 0 0 0 1 1
1 1 1 0 0 0 1 1 0 0 0 0 0 0 1 1 1 1 1 0 1 0 0 1 0 0 0 0 0 0 1 1 1 1 1 1 0 1 0 0
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 1 1
1 1 1 1 1 1 0 1 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1
0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 1 1 0 0 0 1 0 0 0 0 0 0 1 1
1 1 1 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 0 1 0 1 1 0
0 0 0 0 0 0 1 1 1 1 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 0 1 0 0 0 0 0 0 1 1
1 1 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1 1 0 1 0 1 1 0 0 0 0 0 0 0 1 1 1 1 0 0 0 1 0 1
0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 1 1 0 0 0 0 0 0 1 1
1 1 1 0 0 1 0 1 0 0 0 0 0 0 1 1 1 1 0 1 1 1 1 1 0 0 0 0 1 1 1 0 0 1 1 0 1 0 1 0
```


Figure 3.2: Part of the EventStream.dat input file.

```c
struct SW_Data_Packet {
    uint16_t alpha; // Start Constant 0xA1FA
    uint8_t i2c_address; // 3 bits
    uint8_t conf_address; // 4 bits
    uint8_t bank; // 1 bit, A or B
    uint8_t fine_time; // 8 bits, sample number when trigger arrives
    uint32_t coarse_time; // 32 bits
    uint16_t trigger_number; // 16 bits
    uint8_t samples_after_trigger; // 8 bits
    uint8_t look_back_samples; // 8 bits
    uint8_t samples_to_be_read; // 8 bits
    uint8_t starting_sample_number; // 8 bits
    uint8_t number_of_missed_triggers; // 8 bits
    uint8_t state_machine_status; // 8 bits
    uint16_t samples[NUM_SAMPLES][NUM_CHANNELS]; // Variable size?
    // Looks like N samples for 16 channels (so 1 ASIC)
    uint16_t omega; // End Constant 0x0E6A
};
```

Figure 3.3: SW_Data_Packet C struct definition.

```
0  1020 1006 1024 1000 989 975 1007 1033 980 987 992 1000 998 1007 987 1004
1  983 993 1044 983 1003 975 1003 979 1000 1015 995 1011 1037 989 985 1006
2  1026 1018 1028 966 1005 1005 995 999 1016 1007 991 1019 1031 994 990 986
3  1000 1051 1021 1009 995 1026 994 981 994 1012 984 993 984 1015 983 986
4  1032 1025 1044 988 991 947 1010 993 1013 1005 979 1029 1009 996 960 1009
5  1046 1021 977 1007 989 1006 1015 1000 985 983 1000 1019 1010 973 993 960
6  963 1000 979 991 976 1020 1015 1007 1025 1017 1008 986 993 987 1008 1004
7  992 985 994 992 1021 999 1016 1015 1015 990 990 997 962 996 1017 1006
8  1013 985 993 1007 1023 977 990 980 997 988 991 1023 1018 1008 1018 996
9  980 997 1024 978 988 1006 993 1004 998 1013 996 1006 998 981 1002 1025
10  1007 1002 1016 1019 982 991 1027 987 994 999 960 1019 948 982 989 1040
11  1017 994 973 1007 1005 1008 1001 1050 995 1003 1008 1006 1041 973 969 1049
12  1021 1013 1025 1016 986 996 1009 1048 1010 987 1021 983 989 986 1018 981
13  994 989 995 993 975 987 1003 992 977 1020 993 1016 1006 995 990 1003
14  961 998 1032 1035 1007 991 1012 1041 1022 997 991 987 981 999 1019 1025
15  1022 992 975 982 996 1000 998 1010 1014 1036 988 1019 984 1019 1021 996
16  990 1004 1029 978 993 1013 1011 998 1010 1007 994 975 1043 971 991 1001
```

Figure 3.4: Part of the peds.dat input file.

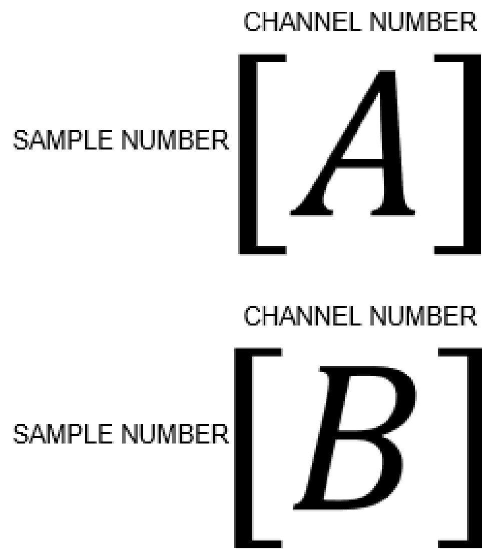

Figure 3.5: The data structure of the peds.dat file.

```
uint16_t all_peds[2][NUM_SAMPLES][NUM_CHANNELS]; // Really 12 bits
```

Figure 3.6: The 3D array representation of the pedestal values.

3.1.3 write_output

The write_output function takes the data packet header information, the specified integral bounds, and the 4 integrals for each of the 16 channels produced by the hardware model to produce an output.txt file that contains all of the aforementioned information. This function has two sub-functions: write_header and write_integrals to handle writing the two sets of information to the output file since they require different formatting. An example output file is shown in Figure 3.7.

```
i2c_address: 0
conf_address: 0
bank: 0
fine_time: 48
coarse_time: 0
trigger_number: 0
samples_after_trigger: 250
look_back_samples: 5
samples_to_be_read: 255
starting_sample_number: 43
number_of_missed_triggers: 0
state_machine_status: 0
0 (-5,5)    11 -5 3 7 -13 3 0 -13 15 2 -12 12 -15 -11 -1 -16
1 (-10,10)    1 -32 13 -19 -12 15 9 -35 23 -6 -15 3 -16 -39 14 -6
2 (-15,15)    -9 -26 15 2 -13 20 -5 -37 9 -12 12 3 -9 -63 23 -16
3 (-20,20)    -9 -16 23 6 -34 7 -13 -44 21 -13 -1 -5 -10 -95 27 13
```

Figure 3.7: An example output.txt file with fabricated integral bounds.

3.1.4 struct_to_json

The struct_to_json function takes the SW_Data_Packet struct instance as an input and produces a JSON output file. Similar to write_output, this function has two corresponding sub-functions: add_to_json and add_samples_to_json. This function set is mainly used to debug the parsing of EventStream.dat but can also be used to provide a data dump of the data packet prior to preprocessing. The JSON format was chosen here because it is a standard data interchange format that is human-readable and uses attribute-value pairs and arrays. A snippet of the packet.json output file is shown in Figure 3.8.

```
{ "alpha": 41466, "i2c_address": 0, "conf_address": 0, "bank": 0, "fine_time": 48, "coarse_tim
e": 0, "trigger_number": 0, "samples_after_trigger": 250, "look_back_samples": 5, "samples_to_
be_read": 255, "starting_sample_number": 43, "number_of_missed_triggers": 0, "state_machine_st
atus": 0, "samples": [ 1023, 989, 1031, 1035, 985, 1037, 982, 976, 992, 983, 1045, 1039, 966,
1033, 1009, 1024, 998, 1043, 1012, 1021, 1017, 1013, 1009, 996, 1013, 991, 998, 980, 971, 989,
 1021, 991, 1006, 1019, 990, 993, 1021, 1017, 989, 977, 1006, 991, 992, 980, 982, 965, 974, 10
27, 977, 1029, 1018, 1005, 997, 992, 1022, 960, 1022, 991, 996, 1034, 964, 1019, 1023, 984, 10
11, 961, 979, 1011, 980, 1000, 1007, 992, 988, 1034, 997, 1018, 994, 1032, 1003, 991, 995, 101
6, 982, 1005, 1013, 1032, 975, 1042, 1018, 1005, 1001, 1013, 1029, 985, 1012, 1000, 977, 966,
987, 995, 957, 992, 1012, 1024, 1008, 1007, 965, 1007, 1015, 1009, 990, 949, 1052, 1012, 983,
989, 1011, 1007, 1006, 1031, 1032, 1002, 964, 953, 1035, 990, 975, 1024, 1009, 954, 997, 990,
1013, 961, 995, 971, 977, 1010, 996, 1026, 1000, 979, 1018, 981, 988, 1000, 1002, 959, 1000, 9
87, 1010, 995, 1005, 1010, 986, 1047, 1001, 982, 1006, 964, 1012, 1000, 1014, 1001, 953, 998,
```


Figure 3.8: A snippet of the packet.json file that is used for debugging and data dumps.

3.2 Hardware Functions

The hardware functions are implemented in the Kernel and are the main focus of the acceleration and optimization efforts. These functions will persist beyond the software pipeline and its corresponding interface functions and be deployed on the flight instrument. The current FPGA data preprocessing effort has two operations: pedestal subtraction and integration. Both of these functions are described in detail below.

3.2.1 ped_subtract

The ped_subtract function is used to remove the pedestal offsets from all of the input samples. This operation is performed on all 256 samples for all 16 channels. The pedestal offsets are specific to each ADC buffer slot. Due to the fact that an event can occur at any time, the starting sample of an event will not necessarily line up with the first ADC buffer slot. And so, special care is needed while indexing the pedestal array in order to subtract the appropriate offset. This is accomplished by starting the sample index at 0 and the pedestal index at the starting sample number, that specifies which ADC buffer slot that sample was stored in. From there, the indices move along their respective arrays, with the pedestal index potentially needing to wrap around once it has reached the end of its array. This indexing scheme and larger operation are depicted in Figure 3.9 and the pseudocode is provided in Figure 3.10.

Figure 3.9: Visual representation of the ped_subtract function.

```
idx = starting sample number;
for i from 0 to number of samples read {
    for j from 0 to NUM_CHANNELS-1 {
        results[i][j] = samples[i][j] - pedestals[bank][idx][j]
    }
    idx += 1;
    if idx == NUM_SAMPLES {
        idx = 0;
    }
}
```

Figure 3.10: Software ped_subtract pseudocode.

3.2.2 integral

The integral function sums over the results of the pedestal subtraction for each of the 16 channels within the specified trigger-relative bounds. The only potential complication within this function is if a specified set of bounds wraps around the data array, in which case special consideration is needed. On further investigation, this wraparound behaviour is not expected since the array being summed over starts with the first sample of the given event and so there will be no earlier samples available. Thus, there will be no need to loop around the ends of the array with our current specified integral types. However, if a new integral type that summed both the pre-signal noise as well as the tail behavior or some similar configuration was deemed to be useful, then this wraparound logic would become relevant once more. The behavior of the two integral configurations–linear and wraparound–are shown in Figure 3.11 and the pseudocode is provided in Figure 3.12.

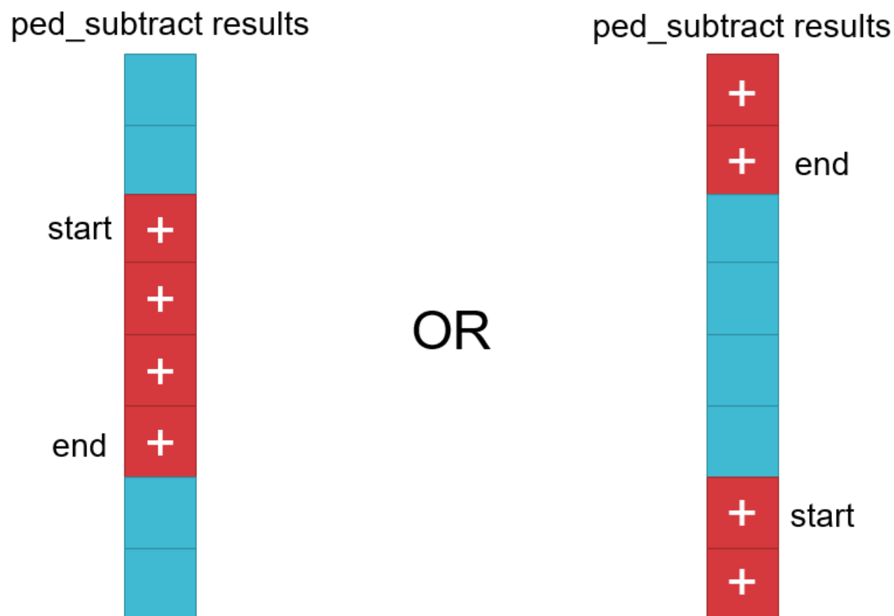

Figure 3.11: Visual representation of the two integral function configurations: linear and wraparound.

```
int32_t integral;

if (end >= start) {
    for i from 0 to NUM_CHANNELS-1 {
        integral = 0;
        for j from start to end {
            integral = integral + samples after ped_subtract[j][i];
        }
        integrals[integral number][i] = integral;
    }
}
else {
    for i from 0 to NUM_CHANNELS-1 {
        integral = 0;
        for j from start to NUM_SAMPLES-1 {
            integral = integral + samples after ped_subtract[j][i];
        }
        for k from 0 to end {
            integral = integral + samples after ped_subtract[k][i];
        }
        integrals[integral number][i] = integral;
    }
}
```

Figure 3.12: Software integral pseudocode.

Chapter 4

HLS Implementation

In order to use HLS to generate an FPGA image, one must first take the software data preprocessing model and split it into Vitis Host and Kernel components. The Host is responsible for creating an OpenCL environment, initializing the inputs to the kernel as memory buffers, running the Kernel, and reading and processing its outputs. This is where the software interface functions described in Chapter 3 are implemented. The Kernel is the equivalent of an FPGA module where the preprocessing algorithms (pedestal subtraction and integration) are implemented. The Host and Kernel communicate over a specified interface. The default interface is AXI, which is a standard ARM bus-protocol.

There are three different types of compilations that can be performed in Vitis: sw_emu, hw_emu, and hw. These are abbreviations for software emulation, hardware emulation, and hardware, respectively. Software emulation is mainly used to verify the functional correctness of the application and typically takes about a minute to compile and run. Hardware emulation provides cycle-accurate results and typically takes about a half hour to compile and run. And finally, a hardware run only compiles the model and produces three files that can then be ran on an appropriate FPGA. These files are app.exe, preprocess.xclbin, and xrt.ini. The generation of these files takes around three hours to complete. All three methods provide compile and link summaries that can be examined using the Vitis Analyzer tool. In the interest of time, most optimization iterations were only compiled and ran using sw_emu and hw_emu. Unless otherwise stated, all reported measurements were taken from hw_emu runs.

The performance goals for this and many other hardware designs are as follows: increase clocking frequency, decrease latency, decrease area, and decrease average power consumption. In this project, we have a set of more specific goals as well. We are aiming for a latency of 2.56 μs, which is the time that it takes to collect 256 10 ns samples. We also want to be able

to support 12 ALPHA ASICs with one FPGA board, which would allow us to process all of either the x or y axis for one of the layers in our detector.

With these performance goals in mind, here are the statistics that we have collected. We are reporting the target and estimated clocking frequencies, which are 300 MHz and 411 MHz, respectively for all emulation runs. We are reporting the worst case latency values in both cycles and seconds. We are reporting the area information in terms of resource usage. The resources that are available to the FPGA are FFs, LUTs, BRAM, DSP, and URAM. We are not currently using the last two resource types. The resource usage is reported in three ways: counts, percent utilization, and an optimistic ASIC count based on the highest percent utilization. Finally, for the hardware runs we are reporting the average power consumption. All of the previously mentioned statistics are reported for the Kernel and not for the combination of the Host and the Kernel.

Before we can discuss the various optimization iterations, a handful of disclaimers. These experiments were conducted on the CloudLab U280 board [9, 10], which is an UltraScale FPGA. The FPGA that we are considering using in the instrument itself is the XC7K325T-2FFG900C part, which is a Kintex-7 FPGA. The decision to compile and run all builds on the U280 was made because that was the part that was supported by Vitis and readily accessible to run on. While the experiments were run on the Ultrascale FPGA, the percent utilization numbers were calculated relative to the Kintex-7 FPGA. Another thing to note is that both the data packets and pedestals are stored in external memory (HBM). Previous discussions have considered placing the pedestals in local memory (BRAM) instead since they are relatively constant. This is still a possibility and would not shift the resource bottleneck onto the BRAM.

4.1 Naive Approach

To implement the naive approach, I started with a Vitis tutorial setup and began modifying it for my use-case [19]. While transforming the combined software model of the data preprocessing into separate Host and Kernel files, we learned a handful of things about the Vitis compiler, specifically what it does and does not understand. One such attribute was a double pointer. We were attempting to transfer an array of char * elements through a char **, but Vitis would not compile our files while also not clearly stating what the issue was.

Another potential issue was packing all functions within a C extern statement. Only the main kernel function (preprocess here) should have this wrapper. The last major issue when developing the naive approach was understanding what memory could be initialized by the Host and communicated to the Kernel. These memory buffers need to be created and then initialized directly using their pointers. The top-level Kernel function preprocess has three inputs and one output. These connections between the Host and the Kernel are shown in Figure 4.1.

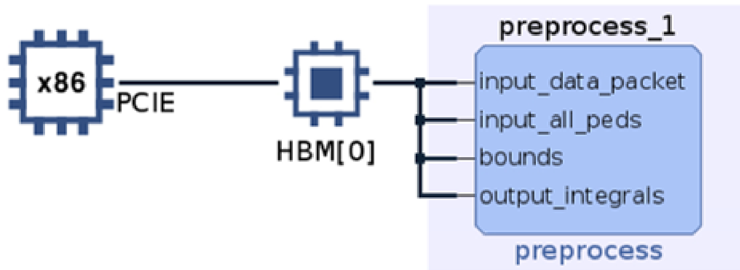

Figure 4.1: Connections between the Vitis Host and Kernel components.

Here we include the naive ped_subtract and integral function implementations so that variations may more easily be discussed. Figures 4.2 and 4.3 show the source code for the naive implementations.

The main data structures within ped_subtract are the SW_Data_Packet struct pointed to by data_packet, the uint16_t 3D array that is pointed to by all_peds, and the int16_t 2D array ped_sub_results, which is a global array stored in local memory. The definition of the SW_Data_Packet struct is shown in Figure 3.3. The functionality of the ped_subtract function is described in Chapter 3.

The main data structures within integral are the SW_Data_Packet struct pointed to by data_packet, the integer values rel_start, rel_end, and integral_num, as well as the output int32_t array pointed to by integrals. The first several lines of this function take the trigger-relative bounds and convert them into the proper indices relative to the starting sample number. The function then conditionally performs either a linear operation or a wraparound operation, depending on whether the calculated start index is less than or greater than the calculated end index. More details on the integral function can be found in Chapter 3.

We performed all three compilation types on the Naive Approach and made several discoveries. First, we discovered a handful of warning messages stating that there was memory

```
int ped_subtract(struct SW_Data_Packet * data_packet, uint16_t *all_peds) {
    int ped_sample_idx = data_packet->starting_sample_number;
    for (int i = 0; i < data_packet->samples_to_be_read + 1; i++) {
        for (int j = 0; j < NUM_CHANNELS; j++) {
            ped_sub_results[i][j] = data_packet->samples[i][j] -
                *(((all_peds + (data_packet->bank)*(NUM_SAMPLES*NUM_CHANNELS)) +
                ped_sample_idx*NUM_CHANNELS) + j);
        }
        ped_sample_idx += 1;
        if (ped_sample_idx == NUM_SAMPLES) {
            ped_sample_idx = 0;
        }
    }
    return 0;
}
```

Figure 4.2: Naive ped_subtract source code.

port contention in ped_subtract and that the integral loop could not be flattened. Another major discovery we made was that because both ped_subtract and integral were imperfect loops, we were unable to obtain any latency estimates from the emulation runs. We were, however, able to obtain an expected layout and resource usage. The expected layout is shown in Figure 4.4 where the dark blue component represents the footprint of the Kernel. The resource usage is reported in Table 4.1.

Table 4.1: Naive hardware emulation area statistics.

FFs	FFs % Util	LUTs	LUTs % Util	BRAM	BRAM % Util	ASICs
14853	3.644	39600	12.14	66	0.4120	8

In order to obtain worst-case latency numbers for the emulation of the Naive Attempt, we added the HLS loop_tripcount pragma that allows you to specify the minimum, average, and maximum number of iterations through a loop. We set the maximum to 256 (NUM_SAMPLES) in order to evaluate the worst-case performance. These latency numbers are reported in Table 4.2.

When we ran the naive approach on a U280 board through the CloudLab, we recorded several important parameters from the run summary. The clocking frequency was reported as 286 MHz which is significantly slower than both our targeted and estimated frequencies.

```
int integral(
    struct SW_Data_Packet * data_packet,
    int rel_start,
    int rel_end,
    int integral_num,
    int32_t * integrals) {
    int start = data_packet->trigger_number + rel_start -
        data_packet->starting_sample_number;
    if (start < 0) {
        start = start + data_packet->samples_to_be_read;
    }
    int end = data_packet->trigger_number + rel_end -
        data_packet->starting_sample_number;
    if (end >= data_packet->samples_to_be_read) {
        end = end - data_packet->samples_to_be_read;
    }
    int32_t integral;

    if (end >= start) {
        for (int i = 0; i < NUM_CHANNELS; i++) {
            integral = 0;
            for (int j = start; j <= end; j++) {
                integral = integral + ped_sub_results[j][i];
            }
            integrals[integral_num*NUM_CHANNELS+i] = integral;
        }
    }
    else {
        for (int i = 0; i < NUM_CHANNELS; i++) {
            integral = 0;
            for (int j = start; j < NUM_SAMPLES; j++) {
                integral = integral + ped_sub_results[j][i];
            }
            for (int k = 0; k <= end; k ++) {
                integral = integral + ped_sub_results[k][i];
            }
            integrals[integral_num*NUM_CHANNELS+i] = integral;
        }
    }
    return 0;
}
```

Figure 4.3: Naive integral source code.

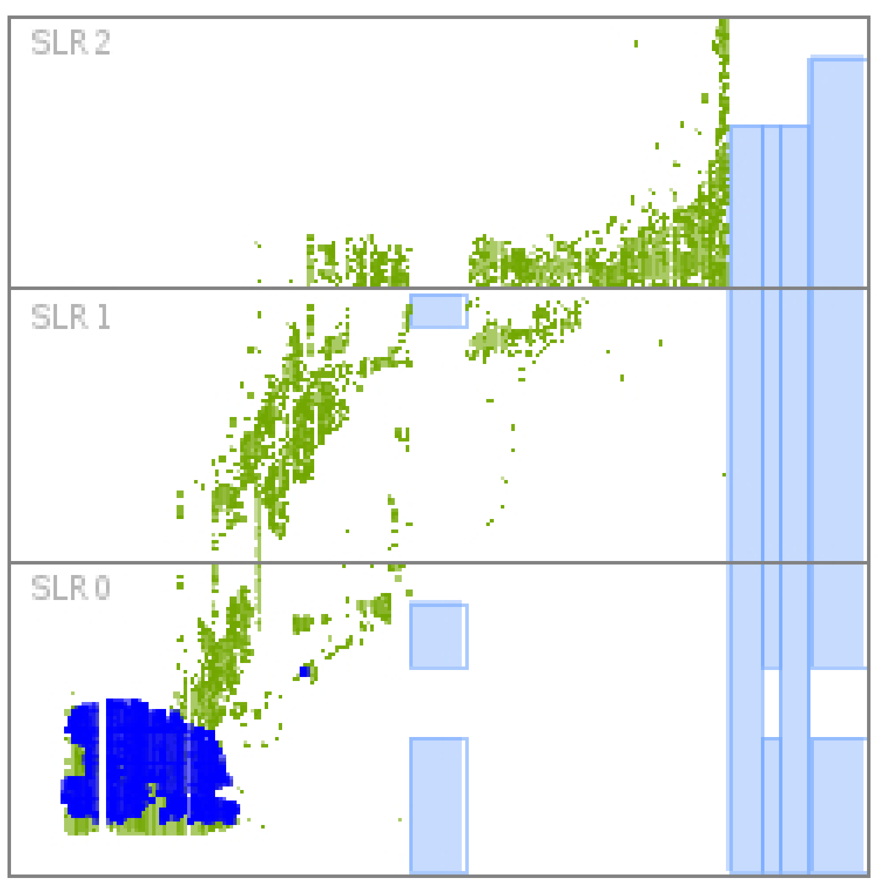

Figure 4.4: The footprint of the U280 with the naive approach. The dark blue highlighted section represents the footprint of the Kernel.

The average power consumption was 10.591 W. The area information is reported in Table 4.3 and the latency information is reported in Table 4.4. As a general trend, the hardware runs produced worse latency but better area statistics than the hardware emulation runs.

4.2 ped_subtract Optimizations

Once we had a fully functional naive approach, we moved on to applying the HLS techniques that we had learned to the ped_subtract function first. Our main focus was to resolve the memory access issue. We then transformed the function into a semi-perfect and perfect loop.

Table 4.2: Naive hardware emulation latency statistics.

Cycles	Time	Times Slower Than Goal
40533	0.135 ms	53

Table 4.3: Naive hardware run area statistics.

FFs	FFs % Util	LUTs	LUTs % Util	BRAM	BRAM % Util	ASICs
21416	5.254	10309	3.161	25	0.1561	19

Table 4.4: Naive hardware run latency statistics.

Cycles	Time	Times Slower Than Goal
84370	0.295 ms	115

In an attempt to fix the memory access issue, we reduced the data_packet field accesses by reading the constant values once and storing them to local variables. This did not resolve the memory warnings and the statistics from hardware emulation remained the same. This implies that the compiler had already seen this opportunity for improvement.

To make ped_subtract into a semi-perfect loop, we moved the pedestal indexing logic into the innermost loop. This modified version is shown in Figure 4.5. For a more complete understanding of the transformation, compare Figures 4.2 and 4.5.

This reconfiguration resolved the memory warnings and produced better area numbers than the naive approach. The area information is reported in Table 4.5 and the latency information is reported in Table 4.6.

Table 4.5: ped_subtract with semi-perfect loop area statistics.

FFs	FFs % Util	LUTs	LUTs % Util	BRAM	BRAM % Util	ASICs
10278	2.522	22753	6.978	52	0.3246	14

In order to verify that semi-perfect and perfect loops have similar performance, we also implemented a perfect loop version of ped_subtract. In doing so, we made an assumption that we will always read the full memory buffer (all 256 samples). This approach also gives us more accurate latency numbers since we are no longer relying on the HLS loop_tripcount pragma. We found that, indeed, the worst-case latency numbers were the same. However, there was a slight change in the area statistics, which are shown in Table 4.7.

```
int ped_subtract(struct SW_Data_Packet * data_packet, uint16_t *all_peds) {
    int ped_sample_idx = data_packet->starting_sample_number;
    uint8_t samples_to_be_read = data_packet->samples_to_be_read;
    uint8_t bank = data_packet->bank;
    for (int i = 0; i < samples_to_be_read + 1; i++) {
        #pragma HLS loop_tripcount min=1 max=256
        for (int j = 0; j < NUM_CHANNELS; j++) {
            ped_sub_results[i][j] = data_packet->samples[i][j] -
                all_peds[bank*NUM_SAMPLES*NUM_CHANNELS +
                ped_sample_idx*NUM_CHANNELS + j];
            if (j==NUM_CHANNELS-1) {
                ped_sample_idx += 1;
                if (ped_sample_idx == NUM_SAMPLES) {
                    ped_sample_idx = 0;
                }
            }
        }
    }
    return 0;
}
```

Figure 4.5: Semi-perfect ped_subtract source code.

Table 4.6: ped_subtract with semi-perfect and perfect loop latency statistics.

Cycles	Time	Times Slower Than Goal
42582	0.142 ms	55

4.3 integral Optimizations

Before diving into the integral optimizations, a disclaimer. Here the integrals are assumed to have variable bounds. If these bounds are found to be constant or it is acceptable to recompile and re-image the FPGA when these bounds change, then a different optimization approach would be taken.

The first optimization iteration involved forcing integral into a perfect loop. By forcing, we mean that as few changes were made as possible while still conforming the requirements of a perfect loop. This forced perfect loop had terrible performance. The iteration interval, which is the number of iterations before a new input can be taken, went from 1 (desired) to

Table 4.7: ped_subtract with perfect loop area statistics.

FFs	FFs % Util	LUTs	LUTs % Util	BRAM	BRAM % Util	ASICs
10261	2.517	22737	6.973	52	0.3246	14

137. This led to a huge increase in latency. There were also memory warnings for the integral ports. On the bright side, the area footprint decreased significantly. These observations are reported in Tables 4.8 and 4.9.

Table 4.8: integral with forced perfect loop area statistics.

FFs	FFs % Util	LUTs	LUTs % Util	BRAM	BRAM % Util	ASICs
4648	1.140	7097	2.176	24	0.1498	45

Table 4.9: integral with forced perfect loop latency statistics.

Cycles	Time	Times Slower Than Goal
2331359	7.770 ms	3035

Our next approach to improving the performance of the integral function involved combining all of the various conditional statements into one condition. This simplification of the logic structure did not affect latency, but did provide a slight decrease in area, as shown in Table 4.10.

Table 4.10: integral with combined logic area statistics.

FFs	FFs % Util	LUTs	LUTs % Util	BRAM	BRAM % Util	ASICs
4517	1.108	6965	2.136	24	0.1498	46

Once again trying to simplify the logic inside of the integral loop, we replaced the combined logic statements with ternary operators. This approach is shown in Figure 4.6. The trigger_number field was replaced by fine_time because we realized that fine_time was the appropriate metric for the sample where the trigger occurred whereas trigger_number specified the number of triggers that had occurred previously. This approach did provide slight improvements to both latency and area. These results are reported in Tables 4.11 and 4.12.

After realizing that simplifying the innermost loop was not going to resolve the memory warning or improve performance much further, we switched tactics to remove a WAR dependency within the function. We used the HLS DEPENDENCE pragma to tell the compiler that

```
int integral(
    struct SW_Data_Packet * data_packet,
    int rel_start,
    int rel_end,
    int integral_num,
    int32_t * integrals) {
    int start = data_packet->fine_time + rel_start -
        data_packet->starting_sample_number;
    if (start < 0) {
        start = start + NUM_SAMPLES - 1;
    }
    int end = data_packet->fine_time + rel_end -
        data_packet->starting_sample_number;
    if (end >= NUM_SAMPLES - 1) {
        end = end - (NUM_SAMPLES - 1);
    }
    int linear = 0;
    if (end >= start) {
        linear = 1;
    }
    for (int i = 0; i < NUM_SAMPLES; i++) {
        for (int j = 0; j < NUM_CHANNELS; j++) {
            int32_t current_integral = integrals[integral_num*NUM_CHANNELS+j];
            integrals[integral_num*NUM_CHANNELS+j] = (i == 0) ? 0 :
                current_integral;
            integrals[integral_num*NUM_CHANNELS+j] = ((i >= start) &&
                (i <= end)) || (!linear && ((i >= start) || (i <= end))) ?
                current_integral + ped_sub_results[i][j] : current_integral;
        }
    }
    return 0;
}
```

Figure 4.6: Ternary operators version of integral source code.

Table 4.11: integral with ternary operators area statistics.

FFs	FFs % Util	LUTs	LUTs % Util	BRAM	BRAM % Util	ASICs
4468	1.096	6987	2.143	24	0.1498	46

Table 4.12: integral with ternary operators latency statistics.

Cycles	Time	Times Slower Than Goal
2314699	7.715 ms	3014

the WAR dependence was a false dependence. Unfortunately, on the first iteration of this approach, we had not successfully resolved the dependence, and so our output was incorrect. This iteration did resolve the memory warnings and provided a significant improvement to latency. The statistics for this functionally incorrect run are shown in Tables 4.13 and 4.14.

Table 4.13: integral with incorrect dependency resolution area statistics.

FFs	FFs % Util	LUTs	LUTs % Util	BRAM	BRAM % Util	ASICs
7140	1.752	6432	1.973	24	0.1498	50

Table 4.14: integral with incorrect dependency resolution latency statistics.

Cycles	Time	Times Slower Than Goal
21499	71.656 μs	28

When we resolved the WAR dependence by adding both a temporary variable and a local memory buffer that is written to the output at the end of execution, functional correctness was restored. Latency was slightly affected while area took a big hit. These results are showed in Tablse 4.15 and 4.16.

Table 4.15: Current approach hardware emulation area statistics.

FFs	FFs % Util	LUTs	LUTs % Util	BRAM	BRAM % Util	ASICs
8242	2.022	18821	5.772	52	0.3246	17

Table 4.16: Current approach hardware emulation latency statistics.

Cycles	Time	Times Slower Than Goal
21382	71.266 μs	28

Since removing the WAR dependence was the last optimization iteration we performed, we also ran that iteration on the U280 board through the CloudLabs. We recorded several important parameters from the run summary. The clocking frequency was reported as 280 MHz which is significantly slower than both our targeted and estimated frequencies. The average power consumption was 8.639 W. The area information is reported in Table 4.17 and the latency information is reported in Table 4.18. As a general trend, the hardware runs produced worse latency but better area statistics than the hardware emulation runs.

Table 4.17: Current approach hardware run area statistics.

FFs	FFs % Util	LUTs	LUTs % Util	BRAM	BRAM % Util	ASICs
14058	3.449	6817	2.091	22	0.1373	28

Table 4.18: Current approach hardware run latency statistics.

Cycles	Time	Times Slower Than Goal
87640	0.313 ms	122

As a final experiment, we separated the ped_subtract logic from the integral logic by simply commenting out the respective function calls. This allowed us to see the area requirements of each function, which are shown in Tables 4.19 and 4.20.

Table 4.19: Current approach hardware run ped_subtract area statistics.

FFs	FFs % Util	LUTs	LUTs % Util	BRAM	BRAM % Util	ASICs
1115	0.2735	926	0.2840	0	0	352

Table 4.20: Current approach hardware run integral area statistics.

FFs	FFs % Util	LUTs	LUTs % Util	BRAM	BRAM % Util	ASICs
4239	1.040	2145	0.6578	8	0.04994	96

In summary, the area numbers that we are seeing look promising, providing us with the ability to optimistically support double the number of ASICs that we had set out to support with the current approach. However, the latency numbers that we are seeing show that further optimizations are needed in order to improve the speed of our Kernel.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The most important conclusions for this thesis are the comparison of the naive and current approaches for HLS implementation and the learnings that were gained throughout this process.

5.1.1 Comparison of Naive and Current Approach

In order to examine the progress of the HLS optimization iterations, here we compare the performance metrics from the naive and current approaches. The naive approach had the number of iterations of the integral loop dependent on the integral bounds. Due to the fact that the naive approach had fewer iterations of the integral loop, the kernel also experienced a lower latency value than the current approach. Though the current approach did experience a higher latency value, the increase relative to the naive approach is not as severe as would be expected from the increase to 256 iterations. While the naive approach outperformed the current approach in terms of latency, the area and power numbers were much better for the current approach. In fact, the area results imply one of the following two options: (1) with additional optimizations, described below, the design can trade additional area for improved latency, or (2) there will be additional area on the FPGA for functions such as centroiding, etc.

5.1.2 Learnings

The first major learning was that the naïve approach took a fair amount of re-arranging to get something Vitis-compatible and functionally correct. This implies that one must be familiar with the Vitis structure and how the Host and Kernel components interact in order to successfully map a software design into an HLS implementation.

The second major learning was that applying HLS optimizations is not straightforward. Throughout the optimization process, the most common issue we encountered was memory port contention. Another issue we encountered was the proper application of HLS pragmas that essentially outsmart the compiler. The user must be extremely careful while applying such pragmas and verify that their proposed solution is still functionally correct with the pragma in place. And finally, another issue with HLS optimization was the trade-off between area and latency. If a technique improved one metric, it generally impaired the other.

The third major learning was that the current approach, with perfect loops for both functions, is just the starting point for HLS optimization. But, all of the above experiments and steps were necessary to produce a design that is configured in such a way that HLS optimizations and pragmas can easily be applied.

5.2 Future Work

The current approach allows for 2.3x the desired number of 12 ASICs to be optimistically supported on a single FPGA. However, the current reported latency is 122x slower than our goal of 2.56 μs. And so, in order to improve the latency / throughput of our design, we must opt for a wider (as opposed to deeper) implementation by parallelizing operations wherever possible. This parallelization will decrease latency by using more computational units and thus more area. And so, the trade-off between latency and area must be explored until there is a satisfactory balance between the depth and width of the design.

The width of the design can be altered using the HLS pragma unroll with various unrolling factors. The unroll factor specifies how many copies of the loop body are desired, which should typically be a power of 2. To implement a fully wide design, the loop would have to be fully unrolled, which can be specified by not setting the factor field in the HLS pragma

unroll command. To implement a design that is less wide, a smaller unroll factor would be used.

The ped_subtract function is trivially parallelizable whereas the integral function could be redesigned as a single progressive sum.

Additional approaches to increased parallelism include vector data types and replicated instances of the Kernel(s). With the exception of the progressive sum (which is an algorithmic improvement) each of the above approaches trades off area for improved speed, so it will be a challenge to fully meet both the area and speed targets simultaneously with the current FPGA. Three options include: (1) a larger FPGA part (e.g., the XC7K480, also in the Kintex-7 family, is 1.5x larger); (2) allotting fewer than 12 ASICs to each FPGA and using a larger number of FPGAs; and/or (3) relaxing the latency requirements.

With the above discussion in mind, below is a list of the future work that we recommend for this project:

- Introduce vector data types of size NUM_CHANNELS or an integral divisor of NUM_CHANNELS.
- Create custom variable types to get more accurate area numbers. Some fields do not require a full 8, 16, 32, etc. bits.
- Combine the pedestal subtraction and integration into one operation.
- Apply bitmasks on both the samples and pedestal values to operate only on the values that are within the integral bounds. This will eliminate the need for conditional execution.
- Transition to a dataflow pipeline, connecting different functions through FIFO buffers to increase throughput.
- Implement memory burst accesses to decrease I/O overhead.
- Investigate whether the OpenCL progressive sum function is possible in a Vitis Kernel.
- Use the HLS unroll pragma to explicitly parallelize functions by creating multiple instances of the module. Experiment with different amounts of unrolling to see the tradeoffs between area and latency.

- Add other algorithms to the FPGA:
  - Centroiding
  - Zero suppression
  - Multi-event detection
  - Data compression
- Use Vitis HLS to create an IP that can be treated as a black box and ported to the target Kintex-7 board.

This thesis has provided an initial investigation into the latency, area, and power required for the preprocessing of gamma-ray transients detected by the Advanced Particle-astrophysics Telescope using HLS.

References

[1] C Altomare, JH Buckley, W Chen, L Di Venere, F Gargano, F Giordano, F Loparco, MN Mazziotta, and D Serini. Simulation of a Compton-pair imaging calorimeter and tracking system for the next generation of MeV gamma-ray telescopes. *Journal of Physics: Conference Series*, 2374(1):012116, 2022.

[2] James Buckley, Samer Alnussirat, Corrado Altomare, Richard G. Bose, Dana L Braun, James H. Buckley, Jeremy Buhler, Eric Burns, Roger D. Chamberlain, Wenlei Chen, Michael L. Cherry, Leonardo Di Venere, Jeffrey Dumonthier, Manel Errando, Stefan Funk, Francesco Giordano, Jonah Hoffman, Zachary Hughes, Dawson J. Huth, Patrick L. Kelly, John F. Krizmanic, Makiko Kuwahara, Francesco Licciulli, Gang Liu, Mario Nicola Mazziotta, John Grant Mitchell, John W. Mitchell, Georgia A. de Nolfo, Riccardo Paoletti, Roberta Pillera, Brian Flint Rauch, Davide Serini, Garry E Simburger, Marion Sudvarg, George Suarez, Teresa Tatoli, Gary S. Varner, Eric A. Wulf, Adrian Zink, and Wolfgang V. Zober. The Advanced Particle-astrophysics Telescope (APT) Project Status. In *Proc. of 37th International Cosmic Ray Conference — PoS(ICRC2021)*, volume 395, pages 655:1–655:9, July 2021.

[3] Anthony M. Cabrera and Roger D. Chamberlain. Designing domain specific computing systems. In *Proc. of IEEE 28th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2020.

[4] Anthony M. Cabrera, Aaron R. Young, Jacob Lambert, Zhili Xiao, Amy An, Seyong Lee, Zheming Jin, Jungwon Kim, Jeremy Buhler, Roger D. Chamberlain, and Jeffrey S. Vetter. Toward evaluating high-level synthesis portability and performance between Intel and Xilinx FPGAs. In *Proc. of 9th International Workshop on OpenCL (IWOCL)*, April 2021.

[5] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[6] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. Transformations of high-level synthesis codes for high-performance computing. *IEEE Transactions on Parallel and Distributed Systems*, 32(5):1014–1029, 2020.

[7] 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 IEEE High-Performance Extreme Computing Conference (HPEC)*, September 2022.

[8] Alexander Finder, Jan-Philipp Witte, and Görschwin Fey. Debugging HDL designs based on functional equivalences with high-level specifications. In *Proc. of 16th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 60–65. IEEE, 2013.

[9] Suranga Handagala, Martin Herbordt, and Miriam Leeser. OCT: The open cloud FPGA testbed. In *Proc. of 31st International Conference on Field Programmable Logic and Applications (FPL)*, 2021.

[10] Suranga Handagala, Miriam Leeser, Kalyani Patle, and Michael Zink. Network attached FPGAs in the Open Cloud Testbed (OCT). In *Proc. of IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2022.

[11] Steven Harris, Roger D. Chamberlain, and Christopher Gill. OpenCL performance on the Intel Heterogeneous Architecture Research Platform. In *Proc. of IEEE High-Performance Extreme Computing Conference (HPEC)*, September 2020.

[12] Maxime Pelcat, Cédric Bourrasset, Luca Maggiani, and François Berry. Design productivity of a high level synthesis compiler versus HDL. In *Proc. of International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 140–147. IEEE, 2016.

[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 (FPT)*, 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, Jeremy Buhler, James H. Buckley, Wenlei Chen, Zachary Hughes, Emily Ramey, Michael L. Cherry, Samer Alnussirat, Ryan Larm, Christofer Chungata, Corrado Altomare, Richard G. Bose, Dana L Braun, Eric Burns, Roger D. Chamberlain, Leonardo Di Venere, Jeffrey Dumonthier, Manel Errando, Stefan Funk, Francesco Giordano, Jonah Hoffman, Dawson J. Huth, Patrick L. Kelly, John F. Krizmanic, Makiko Kuwahara, Francesco Licciulli, Gang Liu, Mario Nicola Mazziotta, John Grant Mitchell, John W. Mitchell, Georgia A. de Nolfo, Riccardo Paoletti, Roberta Pillera, Brian Flint Rauch, Davide Serini, Garry E Simburger, George Suarez, Teresa Tatoli, Gary S. Varner, Eric A. Wulf, Adrian Zink, and Wolfgang V. Zober. A Fast GRB Source Localization Pipeline for the Advanced Particle-astrophysics Telescope. In *Proc. of 37th International Cosmic Ray Conference — PoS(ICRC2021)*, volume 395, pages 588:1–588:9, July 2021.

[16] Marion Sudvarg, Jeremy Buhler, Roger Chamberlain, Chris Gill, and James Buckley. Work in Progress: Real-Time GRB Localization for the Advanced Particle-astrophysics Telescope. In *Proc. of 15th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, pages 57–61, July 2022.

[17] Qi Sun, Tinghuan Chen, Siting Liu, Jianli Chen, Hao Yu, and Bei Yu. Correlated multi-objective multi-fidelity optimization for HLS directives design. *ACM Transactions on Design Automation of Electronic Systems*, 27(4):31:1–31:27, 2022.

[18] Vitis High-Level Synthesis User Guide (UG1399). `https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Getting-Started-with-Vitis-HLS`, 2021.

[19] 2021.1 Vitis Getting Started Tutorial. `https://github.com/Xilinx/Vitis-Tutorials/tree/2021.1/Getting_Started/Vitis`, 2021.

[20] Zhili Xiao, Roger D. Chamberlain, and Anthony M. Cabrera. HLS portability from Intel to Xilinx: A case study. In *Proc. of IEEE High-Performance Extreme Computing Conference (HPEC)*, September 2021.

[21] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In *Proc. of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 409–420. IEEE, 2016.