HLS Taking Flight: Toward Using High-Level Synthesis Techniques in a Space-Borne Instrument

Marion Sudvarg
msudvarg@wustl.edu
McKelvey School of Engineering
Washington Univ. in St. Louis
St. Louis, Missouri, USA

Chenfeng Zhao
chenfeng.zhao@wustl.edu
McKelvey School of Engineering
Washington Univ. in St. Louis
St. Louis, Missouri, USA

Ye Htet
htet.ye@wustl.edu
McKelvey School of Engineering
Washington Univ. in St. Louis
St. Louis, Missouri, USA

Meagan Konst
meagan.konst@wustl.edu
McKelvey School of Engineering
Washington Univ. in St. Louis
St. Louis, Missouri, USA

Thomas Lang
lang.thomas@wustl.edu
McKelvey School of Engineering
Washington Univ. in St. Louis
St. Louis, Missouri, USA

Nick Song
qinzhounick@wustl.edu
McKelvey School of Engineering
Washington Univ. in St. Louis
St. Louis, Missouri, USA

Roger D. Chamberlain
roger@wustl.edu
McKelvey School of Engineering
Washington Univ. in St. Louis
St. Louis, Missouri, USA

Jeremy Buhler
jbuhler@wustl.edu
McKelvey School of Engineering
Washington Univ. in St. Louis
St. Louis, Missouri, USA

James H. Buckley
buckley@wustl.edu
Dept. of Physics
Washington Univ. in St. Louis
St. Louis, Missouri, USA

ABSTRACT

FPGAs are widely deployed on high-energy astrophysics telescopes to preprocess and reduce sensor data read out by front-end electronics. Across instruments, these computational pipelines have similar semantics, sharing common stages such as pedestal subtraction, signal integration, zero-suppression, island detection, and centroiding. However, diverse telescope designs require unique implementations of these algorithms, and the logic is often rewritten from scratch for a new instrument.

As an alternative, High-Level Synthesis (HLS) tools enable these algorithms to be implemented in a high-level language, which eases modifications and enables fast prototyping and deployment. Nonetheless, writing performant HLS code requires augmentation of the code with compiler-specific pragmas. In this work, we illustrate these challenges in the context of the Advanced Particle-astrophysics Telescope (APT), a proposed space-based observatory for gamma-ray sources, and its Antarctic Demonstrator (ADAPT). We implement its front-end algorithms using HLS, demonstrate the use of pragmas to enable optimizations, then explore speed and area tradeoffs, which are especially important given the limited power budget afforded by a satellite instrument. We demonstrate that with HLS, ADAPT will be able to process scintillating tile data from 200 000 gamma-ray events per second.

CCS CONCEPTS

• **Hardware → High-level and register-transfer level synthesis**; • **Applied computing → Astronomy**.

KEYWORDS

astrophysics telescopes, gamma-ray astronomy, hardware synthesis

ACM Reference Format:
Marion Sudvarg, Chenfeng Zhao, Ye Htet, Meagan Konst, Thomas Lang, Nick Song, Roger D. Chamberlain, Jeremy Buhler, and James H. Buckley. 2024. HLS Taking Flight: Toward Using High-Level Synthesis Techniques in a Space-Borne Instrument. In *21st ACM International Conference on Computing Frontiers (CF '24), May 7–9, 2024, Ischia, Italy.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3649153.3649209

1 INTRODUCTION

High-Level Synthesis (HLS) is an approach to expressing hardware designs at a conceptual level higher than that of traditional register-transfer level descriptions. Rather than using a hardware description language such as Verilog or VHDL, the designer expresses the algorithm to be implemented in a high-level language such as C. While the concept is not new, e.g., Streams-C [12] and ROCCC [27] were published two decades ago, the manufacturers of FPGAs have made HLS compilers available commercially only recently. The availability of the tools, however, has not alleviated the challenge of producing performant designs [11, 17, 34, 35, 46].

Currently, when authoring an application using HLS, the language(s) typically used include C, C++, OpenCL, and others; however, invariably they are augmented with additional information to assist the compiler/synthesis suite, often in the form of pragma directives. Sohrabizadeh et al. [35] describe a convolutional neural network implemented in 24 lines of simple HLS code that runs 80× slower than a single thread on an individual processor core. After inserting 28 pragmas, it speeds up 7000×. Of note, many of

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
CF '24, May 7–9, 2024, Ischia, Italy
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0597-7/24/05.
https://doi.org/10.1145/3649153.3649209

these pragma statements are articulating guidance to the tools that is commonly accomplished automatically in the software context, e.g., unrolling loops. This example illustrates the challenge before us: the number of pragmas used to achieve the best performance exceeds the number of lines of code in the initial implementation. In addition, this orders of magnitude performance variability is based not on the core algorithm used to solve the problem, but rather on the low-level specifics of how it is expressed for the HLS compiler/synthesis tool suite.

Nonetheless, HLS enables rapid prototyping and deployment of hardware logic, making it an attractive option for accelerating many applications. Furthermore, it has the potential to make code more readable, maintainable, and portable across functionally different applications of semantically similar algorithms or processes. For example, several high-energy detector instruments used in astrophysics have similar front-end data preprocessing and reduction steps that benefit from acceleration on FPGAs, which can provide substantial improvements in size, weight, and power (SWaP) relative to traditional processors. This is especially important in space-based deployments, which are highly SWaP-constrained.

While instruments differ in size, layout, sampling rates, etc., these algorithms remain largely the same. HLS therefore may lower the barrier toward implementation for such instruments if suitable performance can be achieved. In this paper, we report on our experience using HLS to implement portions of the computational science pipeline onboard APT, a future space-borne, gamma-ray telescope. We describe the initial design efforts, subsequent optimizations, and alternative implementations with an eye toward speed/area tradeoffs that can be exploited in the design. While an initial naïve implementation takes over 15 ms to process a single event, with the addition of several pragma directives and several other HLS-specific optimizations, we can achieve a 3415× speedup without overutilizing the FPGA, enabling processing of over 200 000 gamma-ray events per second.

2 RELATED WORK

A substantial body of recent work has explored performance optimization of arbitrary applications implemented on FPGAs using HLS. Examples include the empirically driven approach of Sanaullah et al. [33]; constraining the application set to a specific domain (e.g., nested loops by Zhong et al. [47], CNNs by Sohrabizadeh et al. [35], or GNNs by Zhao et al. [45]); the use of multi-level intermediate compiler representations by Ye et al. [44]; and the exploitation of an affine type system for compile-time analysis [29]. Much of the above work is motivated by the fact that application developers are required to annotate their code with compiler directives to achieve acceptable performance, and performance variability between an initial implementation and an optimized implementation can be several orders of magnitude [11, 35] — this is true for our application as well.

There has been significant progress in making FPGAs available for space-borne missions in recent years [32, 36], including the RT Kintex [19, 41], SQR Versal [31], and the European NG-MEDIUM [23] and NG-LARGE [20] FPGAs. This has motivated the study of applications and deployment techniques. For example, Lattuada et al. [18] describe an approach to managing data movement automatically. Li et al. [22] developed an FPGA-based CubeSat attitude control system. Menon et al. [25] compare and contrast multispectral sensing algorithms for a CubeSat on traditional processor cores, graphics engines, and FPGAs, showing significant performance benefits for HLS-expressed FPGA implementations.

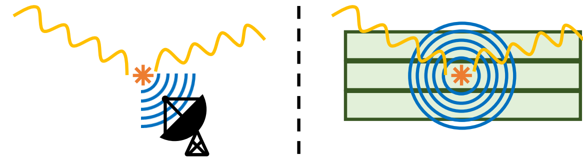

Figure 1: Left: a telescope captures optical Cherenkov radiation from a high-energy particle in the atmosphere. Right: an interaction causes a detector's scintillator to glow.

CubeSat applications are primarily affected by transient bit-flips, rather than long-term radiation damage that might affect a satellite beyond LEO. However, in this work, we are not investigating the suitability or hardening of FPGAs for use in space. Rather, we are focused on assessing whether (or not) HLS techniques are sufficiently mature for use in developing the applications that are commonly required in space-based missions. Janson and Kebschull [16] used HLS to specify a range of computational kernels that are commonly used in detector applications, and they give positive results on both speed and area consumption for the set of kernels that they deploy. Their kernels interact in a dataflow pipeline, which is one of the performance optimizations that we pursue.

3 APPLICATION DESCRIPTION

3.1 Front-End Computation for Telescopes

This paper considers the problem of front-end data reduction and preprocessing for astrophysics instruments. In particular, we consider telescopes like those illustrated in Fig. 1 that measure optical light emitted from high-energy particle interactions, either in the atmosphere (as in imaging atmospheric Cherenkov telescopes such as VERITAS [13, 42] and CTA [8, 10]) or in scintillators in the instrument itself (e.g., COSI [7] and APT [4]).

Many such instruments capture this light using arrays of photomultiplier tubes (PMTs) or silicon photomultipliers (SiPMs) which produce electrical current when struck by an optical photon. Signals from each "pixel" in the array are read by analog waveform digitizers, such as the TARGET [3] or NECTAr0 [9] ASICs. Typically, such ASICs sample continuously into a ring buffer composed of analog memory cells; when triggered by a high-energy interaction, the ASIC digitizes and outputs each sampled charge value. To enable back-end analysis according to the instrument's science goals, a front-end computational pipeline preprocesses and reduces this output data to extract high-level information about the interaction, e.g., position and energy. Common pipeline stages include:

1. Pedestal Subtraction. The capacitive charge pedestal from the ASIC's analog memory cells must be subtracted from its digitized readouts to obtain the true sampled signal values.

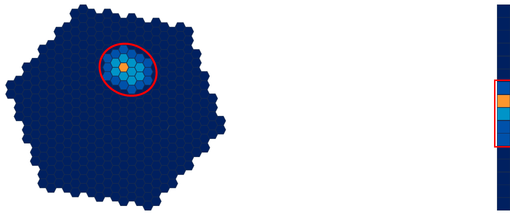

Figure 2: Left: an island of signals detected in a 2D pixel array (e.g., for CTA [5]). Right: an island of signals detected in a 1D pixel array (e.g., for ADAPT [38]).

Figure 3: A rendering of the APT instrument [4].

2. Signal Integration. To infer the number of photons captured by each pixel, shaped waveforms from individual photoelectron outputs, or aggregate outputs across multiple scintillation photons captured by a single pixel, are integrated by summing over the digitized output values. Waveform digitizers typically sample 1–10 ns windows; to capture a significant portion of the arrival curve may require integrating a hundred or more samples per pixel.

3. Zero-Suppression. Sampling noise — e.g., RMS noise from the preamplifiers that shape the PMT or SiPM output prior to capture by the digitizer ASIC [1] or dark counts from SiPMs [30] — may yield positive integrated signal values even for pixels that did not actually capture optical photons. Zero-suppression takes integral values below some configurable threshold and sets them to zero.

4. Island Detection. After zero-suppression, clusters of adjacent pixels with positive integrals are grouped into islands, as illustrated in Fig. 2. These typically correspond to a single interaction.

5. Centroiding. Total signal contribution and the signal-weighted mean over pixel positions in an island constitute a centroid, from which interaction energy and position can be inferred.

3.2 Target Instruments: APT and ADAPT

The Advanced Particle-astrophysics Telescope (APT) [4] (illustrated in Fig. 3) is a concept space-based observatory designed for high-energy gamma-ray and and cosmic-ray observations. It will be deployed in a Sun-Earth Lagrange L_2 orbit, where obscuration of the sky by the earth is minimized, affording it a nearly full-sky field of view. It may achieve an order-of-magnitude improvement in gamma-ray burst (GRB) observational sensitivity compared with the Fermi Gamma-ray Space Telescope, launched in 2008 [24]. To support multi-wavelength and multi-messenger astrophysics [2, 26, 28], APT will detect and localize GRBs in real time [40], directing follow-up instruments to observe GRBs across broad ranges of wavelengths and emission modalities.

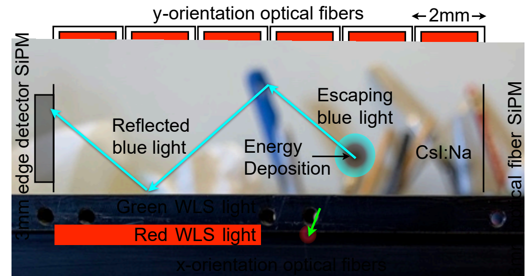

Figure 4: Photons from ADAPT's scintillators are transported by optical fibers to SiPMs forming 1D pixel arrays [4].

To meet these requirements, APT will fly with onboard computational hardware, including FPGAs to accelerate preprocessing and reduction of sensor data. Components of its pipeline, from front-end electronics and ASICs to FPGAs and an analysis CPU, have been discussed in [6, 14, 15, 37–40, 43]. The Antarctic Demonstrator for APT (ADAPT) is a prototype high-altitude balloon mission in advanced development, and is scheduled to fly during the 2025–26 season. ADAPT will demonstrate, at a smaller-scale, the design of the APT instrument as well as its computational capabilities.

ADAPT has 4 layers of sodium-doped cesium iodide (CsI:Na) scintillating crystal tiles. Optical photons produced by energy deposits in the crystals (e.g., from gamma-ray photons as they Compton scatter within the instrument) are captured by perpendicular arrays of optical fibers running across the top and bottom surfaces of each tile and terminated at one end with SiPMs, as illustrated in Fig. 4. These form the 1-dimensional pixel arrays illustrated in the right side of Fig. 2, allowing precise localization of interactions in the crystals along both the x- and y-axes.

Each layer-axis has 225 optical fibers multiplexed into 80 readout channels captured by ALPHA-series waveform digitizer ASICs [38]. The ALPHA has 16 input channels, requiring 5 ALPHAs per layer-axis, or 40 total, to handle the optical fibers. Each channel reads 10 ns samples into a ring buffer of 256 analog memory cells — at 2.56 μs total, this captures up to 98% of the CsI:Na tile's scintillation. When sufficient energy is detected, the ALPHAs are simultaneously triggered, digitizing and reading out to an FPGA.

A single ALPHA data packet consists of an 8-word header, 4096 digitized sample words (16 channels × 256 samples), then a stop word. Of particular significance to the algorithms described in § 4, the header includes (among other data) the following fields:

- `bank`: Which of the two analog memory banks were read out.
- `fine_time`: The sample number (current write position in the ring buffer) when the trigger arrives.
- `starting_sample`: The number of the first sample listed in the output data packet.

In the current planned configuration (Fig. 5), a single FPGA will perform pedestal subtraction, signal integration, zero-suppression, and island detection across outputs from the 5 ALPHAs for a single layer-axis (labeled `X-FPGA` and `Y-FPGA` in the figure). Another

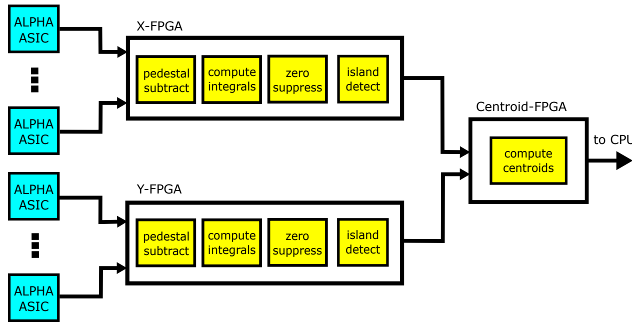

Figure 5: Block diagram of one layer.

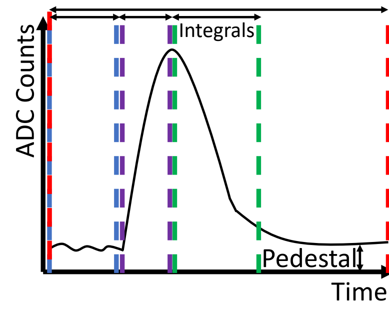

Figure 6: Portions of the waveform captured by integration.

FPGA (labeled `Centroid-FPGA`) will perform centroiding and event building across an entire layer, requiring a total of 12 FPGAs for ADAPT.[1] In the later sections, we synthesize logic for the front-end computational pipeline targeting an embedded Kintex 7 FPGA; it will be used on the ADAPT balloon instrument [38], and its small footprint and low power requirements make it suitably representative for what might fly aboard the future space-based APT.

4 ALGORITHM DESIGN WITH HLS

This section details ADAPT's data preprocessing and reduction algorithms. We present naïve implementations, as might be written in C++ by a developer lacking deeper familiarity with the underlying hardware logic and HLS-specific pragmas and design patterns. We evaluate their area and speed requirements when synthesized for a Kintex 7 FPGA, demonstrating that they are not sufficiently optimized for deployment on a space-borne instrument.

4.1 Naïve Algorithms

1. Pedestal Subtraction. Pedestal values are unique to each analog memory cell; these are stored as 16-bit unsigned values in Block RAM (BRAM) on the FPGA. Each ALPHA has 8192 memory cells (2 banks, 16 channels, 256 samples each), requiring 128 Kib total.

The naïve `ped_subtract` function is shown in Listing 1. It takes pointers to the ALPHA data packet and pedestal value array, plus the ALPHA index `a`. Because the ALPHA begins readout from the `starting_sample`, an appropriate offset into the pedestal array has to be calculated in line ③. The function iterates over samples and channels, subtracting the pedestal from the corresponding sample value, and writes the result into another array for signal integration.

Listing 1: Naïve 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];
} } }
```

2. Signal Integration. Our implementation computes four integrals with configurable bounds to capture different portions of the waveform, e.g., fast and slow components of the signal, initial pedestal values, and the complete waveform, as illustrated in Fig. 6.

The naïve `integrate` function is shown in Listing 2. It iterates over pedestal-subtracted sample values, checks if the sample is within the bounds defined by each integral, and if so, adds it to the corresponding integrated value in an output array. Integral bounds are defined with respect to the triggering time, i.e., when energy was deposited into the detector by an interacting particle. As such, ③ the `fine_time` and `starting_sample` from the packet header are used to calculate a sample number offset. If ④ the offset is negative, this implies that the trigger arrived near enough to the 0-index sample of the ring buffer that the earlier `starting_sample` "wrapped around" to a higher value, so the offset is adjusted accordingly.[2] A time sample is captured by an integral if either ⑪ the (offset) sample number is within the bounds or ⑫ the starting bound is negative and the (offset) sample number exceeds the corresponding index in the ring buffer.

Listing 2: Naïve Implementation of Signal Integration

```
void integrate(Packet *pkt, int *bounds,
               int *integrals, unsigned a) {
  int offst = pkt->fine_time - pkt->starting_sample;
  if (offst < 0) offst += NUM_SAMPLES;
  for (int s = 0; s < NUM_SAMPLES; ++s) {
    int x = s - offst;
    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];
} } } } }
```

3. Zero-Suppression. Our naïve implementation of zero-suppression is shown in Listing 3. The `zero_suppress` function takes pointers to the output integrals and an array of threshold values, one for each integral. Any integral value that is less than the corresponding threshold is set to 0. For ADAPT, only one of the four integrals is likely to be used for island detection and centroiding [38]. However, to remain flexible, we apply zero-suppression to all four integral values; a sufficiently large negative threshold will effectively not suppress values for the corresponding integral.

4. Island Detection. Island detection identifies sequences of adjacent channels with non-zero (i.e., above the zero-suppression

[1] Additional ALPHA ASICs and FPGAs to handle the edge detectors, tail counters, and scintillating fiber trackers described in [38] are outside the scope of this work.

[2] This logic assumes that the ALPHA begins reading out from an earlier sample than the one that was active when the trigger arrived.

Listing 3: Naïve Implementation of Zero-Suppression

```
void zero_suppress(int *integrals, int *thresholds) {
  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;
} } } }
```

threshold) integrated signal values. This is performed across an entire axial array of optical fiber signals, requiring the outputs of zero-suppression from five ALPHA data packets to be merged; our implementation uses only the integrated signal values from the first of the four sets of bounds. In [14], the authors note that for ADAPT, background radiation of the Earth's limb may contribute to significant uncertainty in GRB localization, but that this is mitigated by discarding events with multiple interactions in a single detector layer. This allows a simplified approach that only identifies the number of islands, rather than reporting their regions.

Our naïve implementation of `count_islands` is outlined in Listing 4. A flag, `in_island`, tracks whether it is currently in an island, and a counter, `num_islands` tracks the number of islands identified. In line (8), if the current integrated signal value is positive and the function is not already in an island, then a new island is identified. If, however, (12) the signal value is zero and the function is in an island, (13) the flag is set to false. The function returns the count to be used by centroiding.

Listing 4: Naïve Implementation of Island Detection

```
int count_islands(int *integrals, unsigned int_num) {
  bool in_island = 0;
  int num_islands = 0;
  for (unsigned a = 0; a < NUM_ALPHAS; ++a) {
    for (unsigned c = 0; c < NUM_CHANNELS; ++c) {
      unsigned idx = a * NUM_INTEGRALS * NUM_CHANNELS +
                     int_num * NUM_CHANNELS + c;
      if(integrals[idx] && !in_island) {
        in_island = true;
        ++num_islands;
      }
      else if (!integrals[idx] && in_island) {
        in_island = false;
  } } }
  return num_islands;
}
```

5. Centroiding. Our implementation of island detection returns only an island count and not the boundaries. This is because, as mentioned above, ADAPT may discard events with more than two interactions in a single layer. Our implementation therefore only performs centroiding if a single island has been detected, and centroiding a single island is equivalent to centroiding over the entire array of integrated signal values, as values outside of the island are zero. This simple implementation also has the advantage of predictable timing, as the multiply-accumulate logic does not have to be performed over an array of variable length.

A naïve `centroid` is shown in Listing 5, taking pointers to a `Centroid` data structure and the output integrals, as well as the integral number to consider. The `Centroid` structure encodes two unsigned values: `pos` is the centroid position and `sig` is the total integrated signal. If the number of islands returned by a call to `count_islands` is 1, then `centroid` computes each channel's contribution to the signal-weighted centroid position and total signal.

Listing 5: Naïve Implementation of Centroiding

```
int centroid(Centroid *centr, int *integrals, unsigned int_num) {
  int count = count_islands(integrals, int_num);
  if (count == 1) {
    for (unsigned a = 0; a < NUM_ALPHAS; ++a) {
      for (unsigned c = 0; c < NUM_CHANNELS; ++c) {
        unsigned pos = a * NUM_CHANNELS + c;
        unsigned idx = a * NUM_INTEGRALS * NUM_CHANNELS +
                       int_num * NUM_CHANNELS + c;
        centr->pos += pos * integrals[idx];
        centr->sig += integrals[idx];
    } }
    centr->pos /= centr->sig;
  }
  return count;
}
```

4.2 Synthesis Results

Experimental Setup. We synthesize ADAPT's data preprocessing and reduction algorithms exactly as written in § 4.1 into a single FPGA kernel.[3] We use Vitis HLS version 2021.1, targeting the Kintex-7 KC 705 evaluation platform (XC7K325T) to match the embedded FPGA that will fly aboard the ADAPT balloon instrument [38], applying a conservative 10 ns clock cycle to account for the likely constrained power budget of a future large space-based instrument. We implement a C++ testbench to provide representative input data packets (as described in § 3.2) and pedestal values to each of the 5 ALPHA ASICs, as well as integral bounds and zero-suppression thresholds. The testbench additionally retrieves the integrated signal values and centroid data to verify functional correctness. Inputs and outputs between the testbench and FPGA kernel are implemented as AXI interfaces.

The top-level function is reflected in Listing 6. It calculates offsets into the input pedestal array and output integral array for each ALPHA. It then sequentially calls pedestal subtraction, integration, zero-suppression, and centroiding. Results of pedestal subtraction (to be used by signal integration) are stored in a global array.

Listing 6: Top-Level Function for Naïve Implementation

```
void top(...) {
  for (unsigned alpha = 0; alpha < NUM_ALPHAS; ++alpha) {
    unsigned ped_offst = alpha * 2 * NUM_SAMPLES * NUM_CHANNELS;
    unsigned int_offst = alpha * NUM_INTEGRALS * NUM_CHANNELS;
    ped_subtract(pkts + alpha, peds + ped_offst, alpha);
    integrate(pkts + alpha, bounds, integrals + int_offst, alpha);
    zero_suppress(integrals + int_offst, thresholds);
    centroid_out->count = centroid(centroid_out, integrals, 3);
} }
```

Summary of Results. Synthesis and C/RTL co-simulation results are shown in row **0. Naïve** of Table 1. Latency from co-simulation and the initiation interval (II) reported by synthesis are listed in cycles. The number of BRAM blocks, DSP slices, flip-flops (FFs), and LUTs used (and their percent utilization) are also listed. The naïve implementation incurs a substantial II (>10^6 clock cycles) and a latency of over 15 ms. Given that from a single GRB, *tens of thousands* of gamma-rays may interact with the larger space-based APT *every second*, this implementation is not fast enough for the instrument's intended science goals.

[3] Though centroiding will likely be performed on a separate FPGA, our purpose is to explore the performance and optimization of an HLS-based application. Synthesizing for a single FPGA provides an illustrative proof-of-concept that is easier to follow.

Table 1: Results as reported by Vitis HLS.

Implementation	Latency	II	BRAM	%	DSP	%	FF	%	LUT	%
0. Naïve	1 502 829	1 254 924	92	10	15	1	9 645	2	15 667	7
1. Functional Baseline	1 399 745	1 254 840	48	5	15	1	6 734	1	12 010	5
2. HLS Optimizations	40 056	22 412	48	5	43	43	91 255	22	95 582	46
3. Dataflow Pipelining	440	299	668	75	15	1	44 345	10	88 148	43
4. Consecutive Channel Island Detection	440	299	668	75	15	1	44 417	10	88 417	43
5. Prefix Sum Integrals	443	302	813	91	15	1	36 160	8	63 868	31
6. Sequential ALPHA Execution	6 189	5 071	376	42	15	1	25 256	6	50 949	24

Moreover, the results of signal integration reported by the testbench after C/RTL co-simulation differ from the true results due to corruption of pedestal values during kernel execution.

Functional Baseline. To resolve the functional issues and provide a baseline against which to compare HLS-specific optimizations, we modify the kernel code by using explicit numerical typing, i.e., replacing `int` types with `int`x`_t`, where x is one of 8, 16, or 32 (and similarly for `unsigned`). In addition to using fewer resources, this also resolves the corruption issue reported above. Furthermore, whereas ALPHA packets will arrive over an external interface, pedestal values, integral bounds, and zero-suppression thresholds may already be stored in local BRAM. Therefore, we change these testbench interfaces in the kernel code to `type=bram`.

Results are listed in row **1. Functional Baseline** of Table 1. We observe that resource utilization, II, and simulated latency all decrease. However, the speedup is very minor (only 7%), highlighting the necessity of further performance optimization.

5 PERFORMANCE OPTIMIZATIONS

In this section, we explore and evaluate several HLS-specific performance optimizations. We begin by considering changes to data access patterns and loop ordering to minimize reads across AXI interfaces and to enable pipelining and loop unrolling. We then observe that the ALPHA data packet encodes a Samples×Channels matrix of digitized values that streams into the FPGA, and that pedestal subtraction and integration iterate over the sample index first. By vectorizing across channels, we can implement efficient dataflow pipelining to allow each stage of computation to execute as data becomes available from the prior stage.

5.1 Minor HLS-Specific Optimizations

We begin the optimization process by considering the hardware-level implications of the naïve algorithms as written. We make minor code changes and add pragmas to exploit parallelism and address dependency issues without significant restructuring. Changes are based on the following observations.

Data transfer across AXI interfaces is slow. However, our naïve implementation performs frequent reads and writes across these interfaces. For example, `ped_subtract` reads data from the packet header in each loop iteration, and the output integral array is used to store intermediate results between integration, zero-suppression, island detection, and centroiding.

To optimize access, we add global variables to enable efficient BRAM-based data storage: an array holds results from integration, the output of zero-suppression is stored in a separate array to enable improved pipelining, and a variable stores the centroid information. We add an additional function to the end of our top-level kernel function to copy the local data back over the AXI interfaces to the testbench. We also move the reads from packet headers in `ped_subtract` out of the loop and store the values in local variables.

Some functionality can be pipelined or parallelized. In pedestal subtraction, calculating the offset into the pedestal array based on the trigger time can be moved to the inner loop. While this would introduce unnecessary additional computation in software, this presents opportunities for optimization in hardware synthesis. The change turns pedestal subtraction's loop over samples and channels into a perfectly nested loop, which the synthesis tool automatically flattens. Furthermore, this allows for pipelined execution, which we enable by adding `#pragma HLS PIPELINE II=1`. To avoid an II violation due to limited memory ports, we partition the global `ped_sub_results` array along its channel dimension.

We make a similar change to integration, changing the loop order so that it iterates over channels, then integrals, in a perfectly nested fashion; we also enable pipelining of the flattened loop. Inside this loop, we use a local variable to track the integral value as we iterate over samples with a ternary conditional operator to add pedestal-subtracted sample values if they are inside the integral bounds. After the innermost samples loop, the local value is written back to the global array. This logic allows the inner loop to be automatically unrolled by the synthesis tool. Again, to enable efficient pipelining, we partition the `bounds` BRAM interface variable.

Zero-suppression requires only minor modifications, as it is already implemented with perfectly-nested loops. We use local variables to copy values to and from the global arrays, as well as the BRAM interface for thresholds. The integral value is updated using a ternary operator instead of an `if` statement. These changes allow the outer loop (over integrals) to be pipelined, and the inner loop (over channels) is automatically unrolled.

Besides the use of local variables to access global array members, island detection and centroiding structurally remain largely the same. However, we find that fully unrolling both the outer loop (over ALPHAs) and the inner loop (over channels) via the addition of `#pragma HLS UNROLL` directives improves performance.

Results and Discussion. Results of synthesis and C/RTL co-simulation are listed in row **2. HLS Optimizations** of Table 1. The II is just over 22 thousand cycles (a 56× reduction), and simulated latency is only 400 μs (a 35× speedup). BRAM usage does not increase, but the number of DSP slices, flip-flops, and LUTs increases significantly. However, resource utilization remains under 50% while allowing us

to process ∼ 2500–4500 events per second, even clocked to a conservative 100 MHz. However, this may still not meet the performance goals expected for APT, so we optimize further.

5.2 Dataflow Pipelining

Dataflow programming enables deep task-level pipelining, allowing the parallel execution of functional modules that process streaming data passed via FIFO queues, and minimizing latency due to front-end data ingress. Recently, dataflow programming has been successfully implemented using Intel’s HLS compiler tools for data preprocessing in other detector applications [16]. Using such an architecture, we realize significant performance gains.

Motivation. As illustrated in Fig. 7, an ALPHA ASIC data packet encodes a SAMPLES×CHANNELS matrix of digitized values. Values for all 16 channels for each time sample stream in order to the FPGA before the next time sample arrives. Furthermore, we observe in Listings 1 and 2 that pedestal subtraction and integration both execute over each time sample sequentially; this means that integration can process time samples as soon as pedestal-subtracted values become available, rather than delaying execution until the entire packet has been processed. Similarly, zero-suppression, island detection, and centroiding can process integrated signal values as they become available. To enable deep pipelining across the computational stages, we can execute each function in parallel, connecting them via FIFO queues, and retrieving data using blocking reads.

Vectorized Execution. To realize this design, we make use of vector data types, which the Vitis HLS C++ library provides as a generic type. By vectorizing across the 16 input channels on an ALPHA ASIC, an entire time sample can be retrieved from or written to a FIFO in a single line of code. As each channel is processed equivalently, this simplifies the specification of execution logic. Furthermore, integration (which merely adds values) can naturally be extended to sum over vectorized representations, rather than performing addition on each channel individually.

Implementation. We implement our dataflow architecture according to the structure illustrated in Fig. 7. The logic for each of the five ALPHA ASICs is replicated in parallel. For each ALPHA, an initial function reads time samples, vectorized across channels, from the ALPHA packet; these are then passed via an `hls::stream` FIFO to pedestal subtraction. As pedestal subtraction processes each vector, it passes it to signal integration. Integration produces four integrals per packet (vectorized across channels), which are passed to zero-suppression. The suppressed values across the five ALPHAs are merged and passed to island detection, which itself passes the number of islands and the integrals to centroiding. Centroiding sends the integrals and resulting centroid to functions that write back to the testbench (in the complete implementation, these will send data to the back-end analysis CPU.)

To enable deep pipelining and efficient parallel execution of each pipeline stage, we modify the C++ code that specifies the kernel logic, refactoring and applying several HLS-specific pragmas. We change our testbench and top-level function so that each ALPHA data packet has a dedicated AXI interface, which allows parallel reads from the corresponding memory without contention. The top-level function retrieves the relevant packet header data listed in § 3.2. It then calls a separate `dataflow` function, outlined in Listing 7, which implements the complete pipeline illustrated in Fig. 7. This function defines instances of `hls::stream` variables to provide FIFO queues between its functional blocks and partitions arrays (where necessary) across ALPHAs to provide dedicated access to each parallel instance of `dataflow_alpha`. The `dataflow_alpha` function (also outlined in Listing 7) encapsulates the dataflow functionality for each individual ALPHA, as illustrated in Fig. 7. It is explicitly called 5 times by `dataflow`; since it is decorated with `#pragma HLS FUNCTION_INSTANTIATE variable=alpha`, the compiler synthesizes a distinct copy for each ALPHA.

Listing 7: Dataflow Wrapper Function Implementations

```
1  void dataflow_alpha(..., const uint8_t alpha) {
2    #pragma HLS FUNCTION_INSTANTIATE variable=alpha
3    hls::stream<hls::vector<uint16_t,16>> pkt_samples;
4    #pragma HLS STREAM variable=pkt_samples depth=256
5    ...
6    #pragma HLS DATAFLOW
7    read_samples(samples, pkt_samples);
8    ped_subtract(..., pkt_samples, ped_samples);
9    integrate(..., ped_samples, integrals);
10   zero_suppress(thresholds[alpha], integrals, zeroed_integrals);
11 }
12
13 void dataflow(...) {
14   hls::stream<hls::vector<int32_t,16>> zeroed_integrals[5];
15   #pragma HLS STREAM variable=zeroed_integrals depth=4
16   ...
17   #pragma HLS array_partition variable = peds type=complete dim=1
18   ...
19   #pragma HLS DATAFLOW
20   dataflow_alpha(...,0);
21   ...
22   dataflow_alpha(...,4);
23   merge_integrals(zeroed_integrals, merged_integrals);
24   count_islands(merged_integrals, island_int, num_islands);
25   centroid(island_int, num_islands, centroid_int, local_centroid);
26   write_integrals(centroid_int, output_integrals);
27   write_centroid(local_centroid, centroid_out);
28 }
```

Our implementation of `read_samples` simply loops over time samples in the data packet, reads a complete 16-channel-wide vector of values, and writes them to the `pkt_samples` FIFO. Pedestal subtraction remains largely the same, though it reads a complete vector at a time from the FIFO. Our initial optimization of signal integration in § 5.1 reordered the loops; for our dataflow implementation, we revert back to the loop order in the naïve implementation (Listing 2): for each time sample, it retrieves a complete vector of values from pedestal subtraction. The inner loop over integrals is completely unrolled, allowing each integral to be computed in parallel as time samples are received; an array of temporary integrated values enables efficient pipelining across time samples. At the end of the function, the four resulting vectors are written in order to an output FIFO; these are read by zero-suppression, which writes its results to another FIFO.

The output FIFOs from each ALPHA’s instance of zero-suppression are merged by the `merge_integrals` function in `dataflow`. This passes vectors of integral values in ALPHA, then integral number, order to island detection. While island detection only scans for islands across the specified integral number, it passes all integrals (as well as the number of islands found) to centroiding, as these integral values will be written out. Similarly, centroiding passes all integral

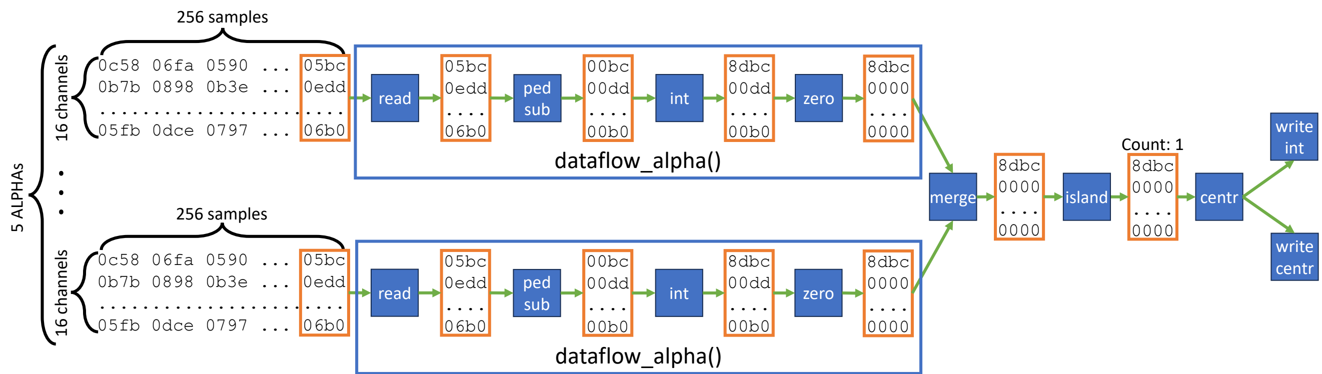

Figure 7: A dataflow architecture enabling task-level parallelism and deep pipelining.

values, but only performs centroiding on the specified integral number. To avoid unnecessary conditional logic, our implementation performs centroiding regardless of the number of islands found, but writes zeroes to the position and signal attributes of the centroid value passed to the output FIFO if the number of islands does not equal 1. Finally, the `write_integrals` and `write_centroid` functions retrieve data from centroiding's output FIFOs and write back to the AXI interfaces.

Results and Discussion. Results of synthesis and co-simulation are listed in row **3. Dataflow Pipelining** of Table 1. While the addition of FIFOs and partitioned arrays increases BRAM usage (though at 75%, it remains underutilized), the FF, LUT, and DSP counts decrease. Most importantly, II decreases 75× to 299 cycles, and latency to 4.4 μs (a 91× speedup) with a conservative 100 MHz clock. This enables processing of at least 227 000 events per second, achieving APT's performance requirements. Furthermore, ALPHAs output 2.56 μs of sampled data, and have a triggering holdoff of at least 3 μs [38], so even faster performance may not be beneficial.

6 EXPLORING ALTERNATIVES

In the previous section, we achieved a sufficiently performant implementation of ADAPT's FPGA-based data reduction and preprocessing pipeline that does not overutilize the FPGA, even if centroiding remains on the same chip. However, our current implementation does not include all necessary functionality; missing components may include edge detector preprocessing, pedestal calibration, and communication with the analysis CPU. Moreover, with the tight SWaP constraints imposed by a space-based instrument, reducing the number of FPGAs may be beneficial. In this section, we explore a few alternative approaches, with a particular eye toward speed and area tradeoffs. Each approach uses the dataflow-based implementation from § 5.2 as a baseline.

6.1 Alternative Algorithms

We begin by independently implementing alternative versions of island detection and signal integration.

Islands as consecutive channels of zero-positive integrated signal values. The implementation of island detection described in § 4.1 iterates over channels using a flag variable to track whether or not it is currently in an island. We now try an alternative iterative approach that scans the integrated signal values for adjacent channels with a zero (post zero-suppression), then a positive value (also accounting for the case where the first channel of the first ALPHA has a positive value). Results are outlined in row **4. Consecutive Channel Island Detection** of Table 1. Latency and II remain the same (not surprising, as island detection did not bottleneck the throughput of our original dataflow implementation). We also observe that BRAM usage remains the same, while flip-flop and LUT utilization increase only slightly (<1% each). This suggests that the alternative algorithm makes little difference, and should not be the preferred methodology. Nonetheless, when performing high-level synthesis, algorithmic changes may have a more substantial impact. These are important to explore, as illustrated by the next example.

Signal integration with prefix sums. In our current implementation, the integrals over four different regions of the incoming signal waveform must be computed independently. Our dataflow implementation executes these in parallel, which minimizes impact on II and latency, but increases resource utilization. We experiment with an alternative approach: by computing a prefix sum over all time samples, the integrals can then be computed as the difference between the values indexed by the integral bounds.

We use a local array of 16-channel-wide vectors to store the prefix sum values, then populate the array as values are read from the FIFO out of pedestal subtraction. Once all values for a single data packet have been read, the function computes each integral. The integral bounds are shifted by the trigger time offset (see line ③ of Listing 2). Similarly to the original implementation, the shift can result in a negative starting index or an ending index that exceeds the size of the array. In this case, the value is shifted appropriately according to the size of the ring buffer. As this would put the starting value ahead of the ending value, a "wraparound" calculation is used:

```
1 integral = (end < start) ?
2   psums[NUM_SAMPLES] - psums[start] + psums[end+1] :
3   psums[end+1] - psums[start];
```

(Note that the prefix sum array, `psums`, is of size 257 with the first element set to 0.)

Synthesis and co-simulation results are shown in row **5. Prefix Sum Integrals** of Table 1. Latency and II increase by just 3 cycles (∼ 1%). Most notably, BRAM usage increases substantially: from 668 to 813 blocks. This is a bit surprising, as the `pedsums` array is

of size 128.5 kib (257 samples × 16 channels × 32 bits per integral value), and a single BRAM block stores 18 kib, suggesting only an extra 8 blocks are necessary, not the 145 that were added. However, flip-flop and LUT usage both decrease significantly. Given that this approach uses 91% of the FPGA's available BRAM, it may not be the preferred implementation. However, it is worth considering in a future implementation where plenty of BRAM remains available while FFs and LUTs become overconstrained.

6.2 Sequential Execution over ALPHA ASICs

As a final alternative, we consider a more substantial tradeoff between speed and area. While our dataflow implementation achieves an event processing rate of over 2×10^5, it also uses 75% of the chip's BRAM and almost half of its LUTs. If a slower event rate is acceptable, we can achieve a lower resource utilization (thereby decreasing the number of FPGAs required) by processing ALPHA packets in sequence, as opposed to using five parallel instantiations of the `dataflow_alpha` function outlined in Listing 7.

Implementation. We modify our baseline dataflow implementation as follows: **First**, because integrals from each ALPHA will be produced sequentially, instead of in parallel, we implement `zeroed_integrals` (line (14) of Listing 7) as a single `hls::stream` FIFO variable, rather than an array. **Second**, the `DATAFLOW` pragma instructs the synthesis tools to implement each contained function call in parallel. As we do not want five separate instantiations of `dataflow_alpha`, we have to call the function sequentially in a loop. However, "the only kind of statements allowed in a canonical dataflow region are variable declarations and function calls."[4] Instead, we wrap the loop in a separate function that takes pointers to each ALPHA data packet interface, using a switch statement to pass the corresponding packet pointer to each call to `dataflow_alpha`. **Third**, we remove the `FUNCTION_INSTANTIATE` pragma from `dataflow_alphas` so that the compiler does not synthesize multiple copies of the function logic.

When we synthesize the kernel in this manner, the tools fail with the following message: "Implementing stream: may cause mismatch if read and write accesses are not in sequential order on port 'zeroed_integrals.' " Despite the fact that, as written, writes are performed by a single function (`zero_suppress`) invoked sequentially across ALPHAs, and reads are performed by a single function (`merge_integrals`) executing in parallel in the `dataflow` function, the tools are unable to synthesize the specified logic. Even passing references to completely separate FIFO queues in each sequential call, then merging all of those queues in `merge_integrals`, fails with the same message.

Results and Discussion. To allow this approach to synthesize, we also have to remove the `DATAFLOW` pragma from `dataflow_alphas`. Results are reported in row **6. Sequential ALPHA Execution** of Table 1. Notice that latency increases 14× and II by 17×, instead of the intended ∼5×, due to the removal of task-level parallelism from the logic invoked by `dataflow_alphas`. Nonetheless, even at a conservative 100 MHz clock, this would enable preprocessing of over 16 thousand events per second while utilizing less than 50% of the FPGA area, possibly enabling a single FPGA to handle an entire layer. While this might not keep up with the event rate associated with an extremely bright burst [21], event processing may already be limited by the data rate from the ALPHA to the FPGA: 16×10^5 data packets require roughly a gigabit.

[4]Taken from a Vitis HLS 2021.1 synthesis warning.

7 CONCLUSIONS

In this work, we have demonstrated the challenges associated with HLS: understanding of the synthesis tools, appropriate selection of programming methodology (e.g., dataflow pipelining), and judicious use of HLS-specific pragmas are all crucial to achieving desired performance. As an illustrative example, we have considered the front-end sensor data reduction and preprocessing pipeline for APT, a future space-based observatory for high-energy gamma-ray and cosmic-ray sources, and synthesized this pipeline for its Antarctic demonstrator (ADAPT), targeting an embedded Kintex 7 FPGA. The performance gains are substantial: compared to a naïve implementation, optimizations provide a 3415× speedup without overutilization, enabling processing of over 2×10^5 gamma-ray events per second with a conservative 100 MHz clock. Furthermore, if energy savings are needed, simple modifications to the HLS-based code allow alternative implementations that may halve the number of FPGAs while still maintaining an II of ∼5000 cycles. These results demonstrate the feasibility of using HLS techniques to synthesize kernels for space-borne instruments.

ACKNOWLEDGMENTS

The authors would like to acknowledge the entire APT Collaboration (see https://adapt.physics.wustl.edu/). Support was provided by NASA award 80NSSC21K1741 and NSF award CNS-1763503.

REFERENCES

[1] C. Aramo, E. Bissaldi, M. Bitossi, et al. 2023. A SiPM multichannel ASIC for high Resolution Cherenkov Telescopes (SMART) developed for the pSCT camera telescope. *Nucl. Instrum. Methods Phys. Res. A* 1047 (2023), 167839. https://doi.org/10.1016/j.nima.2022.167839

[2] Imre Bartos and Marek Kowalski. 2017. *Multimessenger Astronomy.* IOP Publishing. https://doi.org/10.1088/978-0-7503-1369-8

[3] K Bechtol, S Funk, A Okumura, LL Ruckman, A Simons, H Tajima, J Vandenbroucke, and GS Varner. 2012. TARGET: A multi-channel digitizer chip for very-high-energy gamma-ray telescopes. *Astroparticle Physics* 36, 1 (2012), 156–165. https://doi.org/10.1016/j.astropartphys.2012.05.016

[4] James Buckley et al. 2021. The Advanced Particle-astrophysics Telescope (APT) Project Status. In *Proc. of 37th Int'l Cosmic Ray Conference*, Vol. 395. Sissa Medialab, 655:1–655:9. https://doi.org/10.22323/1.395.0655

[5] S. Caroff, P. Aubert, E. Garcia, G. Maurin, V. Pollet, and T. Vuillaume. 2023. The Real Time Analysis Framework of the Cherenkov Telescope Array's Large-Sized Telescope. In *Proc. of 38th International Cosmic Ray Conference*, Vol. 444. Sissa Medialab, 616:1–616:9. https://doi.org/10.22323/1.444.0616

[6] Wenlei Chen, James Buckley, et al. 2023. Simulation of the instrument performance of the Antarctic Demonstrator for the Advanced Particle-astrophysics Telescope in the presence of the MeV background. In *Proc. of 38th Int'l Cosmic Ray Conference*, Vol. 444. Sissa Medialab, 841:1–841:9. https://doi.org/10.22323/1.444.0841

[7] Jeng-Lun Chiu, Steven E Boggs, Carolyn A Kierans, Alex Lowell, Clio Sleator, John A Tomsick, Andreas Zoglauer, Mark Amman, Hsiang-Kuang Chang, Che-Yen Chu, et al. 2017. The Compton Spectrometer and Imager (COSI). In *Proc. of 35th International Cosmic Ray Conference*, Vol. 301. Sissa Medialab, 796:1–796:8.

[8] CTA Consortium, M Actis, G Agnetta, F Aharonian, A Akhperjanian, J Aleksić, E Aliu, D Allan, I Allekotte, F Antico, et al. 2011. Design concepts for the Cherenkov Telescope Array CTA: an advanced facility for ground-based high-energy gamma-ray astronomy. *Experimental Astronomy* 32 (2011), 193–316. https://doi.org/10.1007/s10686-011-9247-0

[9] E. Delagnes et al. 2011. NECTAr0, a new high speed digitizer ASIC for the Cherenkov Telescope Array. In *IEEE Nuclear Science Symposium Conference Record.* IEEE, 1457–1462. https://doi.org/10.1109/NSSMIC.2011.6154348

[10] G Dubus, JL Contreras, S Funk, Y Gallant, T Hassan, J Hinton, Y Inoue, Jürgen Knödlseder, P Martin, N Mirabal, et al. 2013. Surveys with the Cherenkov telescope array. *Astroparticle Physics* 43 (2013), 317–330. https://doi.org/10.1016/j.astropartphys.2012.05.020

[11] Clayton J Faber, Steven D Harris, Zhili Xiao, Roger D Chamberlain, and Anthony M Cabrera. 2022. Challenges Designing for FPGAs Using High-Level Synthesis. In *Proc. of High Performance Extreme Computing Conference*. IEEE, 7 pages. https://doi.org/10.1109/HPEC55821.2022.9926398

[12] Maya Gokhale, Jan Stone, Jeff Arnold, and Mirek Kalinowski. 2000. Stream-oriented FPGA computing in the Streams-C high level language. In *Proc. of Symposium on Field-programmable Custom Computing Machines*. IEEE, 49–56. https://doi.org/10.1109/FPGA.2000.903392

[13] J Holder, VA Acciari, E Aliu, T Arlen, M Beilicke, W Benbow, SM Bradbury, JH Buckley, V Bugaev, Y Butt, et al. 2008. Status of the VERITAS Observatory. In *AIP Conference Proceedings*, Vol. 1085. American Institute of Physics, 657–660. https://doi.org/10.1063/1.3076760

[14] Y. Htet, M. Sudvarg, J. Buhler, R.D. Chamberlain, and J.H. Buckley. 2023. Localization of Gamma-ray Bursts in a Balloon-Borne Telescope. In *Proc. of Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W)*. ACM, 395–398. https://doi.org/10.1145/3624062.3624107

[15] Ye Htet, Marion Sudvarg, Jeremy Buhler, Roger Chamberlain, Wenlei Chen, James H. Buckley, et al. 2023. Prompt and Accurate GRB Source Localization Aboard the Advanced Particle Astrophysics Telescope (APT) and its Antarctic Demonstrator (ADAPT). In *Proc. of 38th Int'l Cosmic Ray Conference*, Vol. 444. Sissa Medialab, 956:1–956:9. https://doi.org/10.22323/1.444.0956

[16] Thomas Janson and Udo Kebschull. 2023. Data pre-processing with high-level-synthesis and dataflow programming using HLS C++ dataflow template library. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 1045 (2023), 167594. https://doi.org/10.1016/j.nima.2022.167594

[17] Sakari Lahti, Panu Sjövall, Jarno Vanne, and Timo D Hämäläinen. 2018. Are we there yet? A study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (2018), 898–911. https://doi.org/10.1109/TCAD.2018.2834439

[18] Marco Lattuada, Fabrizio Ferrandi, and Maxime Perrotin. 2017. Data transfers analysis in computer assisted design flow of FPGA accelerators for aerospace systems. *IEEE Transactions on Multi-Scale Computing Systems* 4, 1 (2017), 3–16. https://doi.org/10.1109/TMSCS.2017.2699647

[19] David S Lee, Gregory R Allen, Gary Swift, Matthew Cannon, Michael Wirthlin, Jeffrey S George, Rokutaro Koga, and Kangsen Huey. 2015. Single-event characterization of the 20 nm Xilinx Kintex Ultrascale field-programmable gate array under heavy ion irradiation. In *Proc. of IEEE Radiation Effects Data Workshop*. IEEE, 6 pages. https://doi.org/10.1109/REDW.2015.7336736

[20] Vasileios Leon, Ioannis Stamoulias, George Lentaris, Dimitrios Soudris, David Gonzalez-Arjona, Ruben Domingo, David Merodio Codinachs, and Isabelle Conway. 2021. Development and testing on the European space-grade BRAVE FPGAs: Evaluation of NG-large using high-performance DSP benchmarks. *IEEE Access* 9 (2021), 131877–131892. https://doi.org/10.1109/ACCESS.2021.3114502

[21] S Lesage, P Veres, MS Briggs, A Goldstein, D Kocevski, E Burns, CA Wilson-Hodge, PN Bhat, D Huppenkothen, CL Fryer, et al. 2023. Fermi-GBM discovery of GRB 221009A: An extraordinarily bright GRB from onset to afterglow. *The Astrophysical Journal Letters* 952, 2 (2023), L42.

[22] Junquan Li, Mark Post, and Regina Lee. 2015. FPGA hardware nonlinear control design for modular CubeSat attitude control system. In *Proc. of IEEE Aerospace Conference*. IEEE, 15 pages. https://doi.org/10.1109/AERO.2015.7119084

[23] K. Maragos, V. Leon, G. Lentaris, D. Soudris, D. Gonzalez-Arjona, R. Domingo, A. Pastor, D.M. Codinachs, and I. Conway. 2018. Evaluation methodology and reconfiguration tests on the new European NG-MEDIUM FPGA. In *Proc. of NASA/ESA Conference on Adaptive Hardware and Systems*. IEEE, 127–134. https://doi.org/10.1109/AHS.2018.8541492

[24] Charles Meegan, Giselher Lichti, P. N. Bhat, et al. 2009. The Fermi Gamma-Ray Burst Monitor. *The Astrophysical Journal* 702, 1 (Aug. 2009), 791–804. https://doi.org/10.1088/0004-637x/702/1/791

[25] Vivek V Menon, Saquib A Siddiqui, Sanil Rao, Andrew Schmidt, Matthew French, Ved Chirayath, and Alan Li. 2021. Design and performance evaluation of multi-spectral sensing algorithms on CPU, GPU, and FPGA. In *Proc. of IEEE Aerospace Conference*. IEEE, 9 pages. https://doi.org/10.1109/AERO50100.2021.9438307

[26] Péter Mészáros, Derek B Fox, Chad Hanna, and Kohta Murase. 2019. Multi-messenger astrophysics. *Nature Reviews Physics* 1, 10 (2019), 585–599.

[27] WA Najjar, W Bohm, BA Draper, J Hammes, R Rinker, JR Beveridge, M Chawathe, and C Ross. 2003. High-level language abstraction for reconfigurable computing. *Computer* 36, 8 (2003), 63–69. https://doi.org/10.1109/MC.2003.1220583

[28] Andrii Neronov. 2019. Introduction to multi-messenger astronomy. In *Journal of Physics: Conference Series*, Vol. 1263. IOP Publishing, 012001.

[29] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable accelerator design with time-sensitive affine types. In *Proc. of 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 393–407. https://doi.org/10.1145/3395657

[30] Adam Nepomuk Otte, Distefano Garcia, Thanh Nguyen, and Dhruv Purushotham. 2017. Characterization of three high efficiency and blue sensitive silicon photomultipliers. *Nucl. Instrum. Methods Phys. Res. A* 846 (Feb. 2017), 106–125. https://doi.org/10.1016/j.nima.2016.09.053

[31] N. Perryman, C. Wilson, and A. George. 2023. Evaluation of Xilinx Versal Architecture for Next-Gen Edge Computing in Space. In *Proc. of IEEE Aerospace Conference*. IEEE, 11 pages. https://doi.org/10.1109/AERO55745.2023.10115906

[32] Sebastian Sabogal and Alan George. 2021. A Methodology for Evaluating and Analyzing FPGA-Accelerated, Deep-Learning Applications for Onboard Space Processing. In *Proc. of IEEE Space Computing Conference*. IEEE, 143–154. https://doi.org/10.1109/SCC49971.2021.00022

[33] A. Sanaullah, R. Patel, and M. Herbordt. 2018. An empirically guided optimization framework for FPGA OpenCL. In *Proc. of International Conference on Field-Programmable Technology*. IEEE, 46–53. https://doi.org/10.1109/FPT.2018.00018

[34] Pedro Filipe Silva, João Bispo, and Nuno Paulino. 2021. FPGAs as General-Purpose Accelerators for Non-Experts via HLS: The Graph Analysis Example. In *Proc. of International Conference on Field-Programmable Technology*. IEEE, 4 pages. https://doi.org/10.1109/ICFPT52863.2021.9609832

[35] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators. *ACM Transactions on Design Automation of Electronic Systems* 27, 4 (2022), 32:1–32:27. https://doi.org/10.1145/3494534

[36] Luca Sterpone, Sarah Azimi, and Corrado De Sio. 2023. A Framework for Uniformly Analyze and Mitigate Radiation-effects on FPGAs for Aerospace. In *Proc. of 20th ACM International Conference on Computing Frontiers*. ACM, 257–262. https://doi.org/10.1145/3587135.3592768

[37] Marion Sudvarg et al. 2021. A Fast GRB Source Localization Pipeline for the Advanced Particle-astrophysics Telescope. In *Proc. of 37th Int'l Cosmic Ray Conference*, Vol. 395. Sissa Medialab, 588:1–588:9. https://doi.org/10.22323/1.395.0588

[38] Marion Sudvarg et al. 2023. Front-End Computational Modeling and Design for the Antarctic Demonstrator for the Advanced Particle-astrophysics Telescope. In *Proc. of 38th International Cosmic Ray Conference*, Vol. 444. Sissa Medialab, 764:1–764:9. https://doi.org/10.22323/1.444.0764

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

[40] Marion Sudvarg, Jeremy Buhler, Roger D. Chamberlain, Chris Gill, James Buckley, and Wenlei Chen. 2023. Parameterized Workload Adaptation for Fork-Join Tasks with Dynamic Workloads and Deadlines. In *Proc. of IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 232–242. https://doi.org/10.1109/RTCSA58653.2023.00035

[41] Louis van Harten, Roel Jordans, and Hamid Pourshaghaghi. 2017. 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*. IEEE, 299–306. https://doi.org/10.1109/DSD.2017.45

[42] TC Weekes, H Badran, SD Biller, I Bond, S Bradbury, J Buckley, D Carter-Lewis, M Catanese, S Criswell, W Cui, et al. 2002. VERITAS: the very energetic radiation imaging telescope array system. *Astroparticle Physics* 17, 2 (2002), 221–243. https://doi.org/10.1016/S0927-6505(01)00152-9

[43] Jacob Wheelock, William Kanu, Marion Sudvarg, et al. 2021. Supporting Multi-messenger Astrophysics with Fast Gamma-ray Burst Localization. In *Proc. of IEEE/ACM HPC for Urgent Decision Making Workshop*. IEEE, 8 pages. https://doi.org/10.1109/UrgentHPC54802.2021.00008

[44] H. Ye, H. Jun, H. Jeong, S. Neuendorffer, and D. Chen. 2022. ScaleHLS: a scalable high-level synthesis framework with multi-level transformations and optimizations. In *Proc. of 59th ACM/IEEE Design Automation Conference*. ACM, New York, NY, USA, 1355–1358. https://doi.org/10.1145/3489517.3530631

[45] Chenfeng Zhao, Zehao Dong, Yixin Chen, Xuan Zhang, and Roger D Chamberlain. 2023. GNNHLS: Evaluating Graph Neural Network Inference via High-Level Synthesis. In *Proc. of 41st International Conference on Computer Design*. IEEE, 574–577. https://doi.org/10.1109/ICCD58817.2023.00092

[46] Chenfeng Zhao, Clayton J. Faber, Roger D. Chamberlain, and Xuan Zhang. 2024. HLPerf: Demystifying the Performance of HLS-based Graph Neural Networks with Dataflow Architectures. *ACM Transactions on Reconfigurable Technology and Systems* (2024). https://doi.org/10.1145/3655627

[47] Guanwen Zhong, Vanchinathan Venkataramani, Yun Liang, Tulika Mitra, and Smail Niar. 2014. Design space exploration of multiple loops on FPGAs using high level synthesis. In *Proc. of IEEE 32nd International Conference on Computer Design*. IEEE, 456–463. https://doi.org/10.1109/ICCD.2014.6974719

A ARTIFACT APPENDIX

A.1 Abstract

This artifact contains the kernel and test bench source code for high-level synthesis (HLS) of ADAPT's data preprocessing and reduction algorithms. Code is written in C++ for compilation in AMD XILINX Vitis HLS version 2021.1, targeting the Kintex-7 KC 705 evaluation platform (XC7K325T).

This artifact reproduces the results in Table 1. Users need the free Vitis HLS software to emulate the target FPGA. A physical FPGA is **not** required.

A.2 Author Information

For questions about this artifact, please contact Marion Sudvarg, McKelvey School of Engineering, Washington University in St. Louis, St. Louis, Missouri, USA, msudvarg@wustl.edu.

A.3 Artifact check-list (meta-information)

- **Algorithm:** We present implementations of data pre-processing algorithms (Listings 1–5) for high-energy particle telescopes.
- **Program:** C++ implementations of test-bench and HLS kernels.
- **Data set:** Sample ALPHA ASIC data packet and pedestal files.
- **Run-time environment:** AMD XILINX Vitis HLS 2021.1. Can be installed on Windows 10, 11, or Linux with desktop environment.
- **Metrics:** FPGA kernel speed and area.
- **Output:** Synthesis reports with data in Table 1.
- **Approximate disk space required:** 50GB to install Vitis HLS.
- **Approximate preparation time:** 1–2 hours to install Vitis HLS.
- **Approximate experiment time:** 1 hour.
- **Publicly available:** https://doi.org/10.7936/6RXS-103658
- **Code and data licenses:** MIT.

A.4 Description

A.4.1 How to access. This artifact is available on GitHub and an archival repository through the Washington University in St. Louis libraries:
https://doi.org/10.7936/6RXS-103658
https://data.library.wustl.edu/record/103658
https://github.com/McKelvey-Engineering-CSE/adapt_fpga/tree/computing-frontiers-2024

A.4.2 Hardware dependencies. No special hardware needed. Only a laptop or desktop running Windows 10, 11, or Linux with a desktop environment that can support AMD XILINX Vitis HLS 2021.1.

The Vitis HLS installation requires around 50GB of free space.

Synthesis and co-simulation of all versions of the kernel pipeline requires around 1 hour on a Linux machine with 8 cores and 16GB of memory. However, 4 cores and 4GB of memory are sufficient.

A.4.3 Software dependencies. AMD XILINX Vitis HLS 2021.1.

A.4.4 Data sets. ALPHA ASIC sample data packet, `EventStream.dat`, and pedestal value, `peds.dat`, are included.

A.5 Installation

Detailed instructions and screenshots are in the repository's 'README.pdf.'

A.5.1 Install Vitis HLS. Install AMD XILINX Vitis HLS 2021.1 per these instructions: https://docs.xilinx.com/r/2021.1-English/ug1393-vitis-application-acceleration/Installing-the-Vitis-Software-Platform

Download the "Vitis Core Development Kit - 2021.1 Full Product Installation" installer: https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vitis/archive-vitis.html

A.5.2 Create Project. Launch Vitis HLS, create a new project, then add the design files: `preprocess.cpp`, `preprocess.h`.

Select `preprocess` as the top function. Next, add testbench files: `host_hls.cpp`, `preprocess.h`, `EventStream.dat`, `peds.dat`.

On the next screen, set a 10 ns clock period, then select a part. Search by "Boards" and select "Kintex-7 KC705 Evaluation Platform." Finally, click "Finish." Your evaluation environment will launch.

A.6 Experiment workflow

We compare the reported performance of the seven different implementations of our data pre-processing pipeline listed in Table 1. The II (initiation interval in cycles) and resource utilization (BRAM, DSP, FF, and LUT) metrics are taken from the Vitis HLS C synthesis report. Latency, in cycles, is taken from the Vitis HLS C/RTL co-simulation report.

The repository contains a copy of the testbench and kernel files for each implementation. For example, *(i)* `1-preprocess.cpp` is the HLS code for the kernel corresponding to implementation **1. Functional Baseline**; *(ii)* `1-host_hls.cpp` is the C++ testbench corresponding to **1. Functional Baseline**; and *(iii)* `1-preprocess.h` is the common header file for the two. We also provide a header, `filepath.h`, that defines paths to the input data and output result files used in simulation. Modify the following line to reflect the actual path to the repository on your evaluation system:

```
1 const std::string path_to_repo = "/home/yourusername/adapt_fpga/";
```

For example, these steps evaluate **1. Functional Baseline**:

(1) Modify `filepath.h` as appropriate.
(2) Launch Vitis HLS: `vitis_hls`
(3) Copy corresponding project files into project:

```
1 cp 1-preprocess.cpp preprocess.cpp
2 cp 1-preprocess.h preprocess.h
3 cp 1-host_hls.cpp host_hls.cpp
```

(4) Run C Simulation (typically takes < 1 minute).
(5) Run C Synthesis (typically takes 3–5 minutes).
(6) Run C/RTL Cosimulation (typically takes 5–10 minutes).

A.7 Evaluation and expected results

A.7.1 C Simulation. This process compiles the C++ testbench and kernel code into a software binary and runs it. At the end of a run, the file `output.txt` will be produced in the repository directory pointed to by `filepath.h`. It contains debugging output information from the pre-processing pipeline. The correct output that can be compared with `diff` can be found at `output_five_centroiding.txt`.

A.7.2 C Synthesis. This step generates and analyzes the FPGA kernel. At the end of a run, Vitis HLS will display a report called "Synthesis Summary." This contains the II, BRAM, DSP, FF, and LUT values shown in Table 1. The report file is generated in `/(solution path)/syn/report/csynth.rpt` where `(solution path)` is the path to the Vitis HLS solution specified when creating the project. The file has additional details, including the resource utilization percentages in Table 1. We have provided the corresponding report for each implementation in the repository. For example, `1-csynth.rpt` corresponds to implementation **1. Functional Baseline**.

A.7.3 C/RTL Co-Simulation. This step compiles a software binary for the testbench and runs it simultaneously with the synthesized kernel in an emulation environment. At the end of a run, Vitis HLS will display a report called "Co-Simulation Report" which contains the latency value shown in Table 1. The file is `/(solution path)/sim/report/preprocess_cosim.rpt`. We have provided the corresponding report for each implementation in the repository. For example, `1-preprocess_cosim.rpt` corresponds to implementation **1. Functional Baseline**.

This step also creates an `output.txt` file similarly to C Synthesis. It can also be compared to `output_five_centroiding.txt`. As we have noted in the paper, the output produced by the **0. Naïve** implementation is incorrect, and is instead expected to match the file `output_naive.txt`.