Optimization and Hardware Acceleration of Event-Based Optical Flow for Real-Time Processing and Compression on Embedded Platforms

by

Daniel Charles Stumpp

B.S. Electrical Engineering, University of Pittsburgh, 2020

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2022

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Daniel Charles Stumpp

It was defended on

April 7, 2022

and approved by

Dr. Ryad Benosman, PhD., Professor, Department of Electrical and Computer Engineering

Dr. Rajkumar Kubendran, PhD., Professor, Department of Electrical and Computer Engineering

Thesis Advisor: Dr. Alan George, PhD., Department Chair and Mickle Chair Professor, Department of Electrical and Computer Engineering

Optimization and Hardware Acceleration of Event-Based Optical Flow for Real-Time Processing and Compression on Embedded Platforms

Daniel Charles Stumpp, M.S.

University of Pittsburgh, 2022

Event-based vision sensors produce asynchronous event streams with high temporal resolution based on changes in the visual scene. The properties of these sensors allow for accurate and fast calculation of optical flow as events are generated. Existing solutions for calculating optical flow from event data either fail to capture the true direction of motion due to the aperture problem, do not use the high temporal resolution of the sensor, or are too computationally expensive to be run in real time on low-power embedded platforms. In this research, software optimization of the existing ARMS (Aperture Robust Multi-Scale flow) algorithm is performed. The new optimized software version (fARMS) significantly improves throughput on a traditional CPU. Further, we present hARMS, a hardware realization of the fARMS algorithm allowing for real-time computation of true flow on low-power, embedded platforms. The proposed hARMS architecture targets hybrid system-on-chip (SoC) devices and was designed to maximize configurability and throughput. The hardware architecture and fARMS algorithm were developed with asynchronous neuromorphic processing in mind, abandoning the common use of an event frame and instead operating using only a small history of relevant events, allowing latency to scale independently of the sensor resolution. This change in processing paradigm improved the estimation of flow directions by up to 73% compared to the original ARMS algorithm and yielded a demonstrated hARMS throughput of up to 1.21 Mevent/s on the benchmark configuration selected. This throughput enables real-time performance and makes it the fastest known realization of aperture-robust, event-based optical flow to date. Finally, the fast event-based optical flow enabled by the fARMS and hARMS designs is leveraged for use in a novel flow-based event-stream compression algorithm. This compression method enables communication across low-bandwidth mediums, while allowing for accurate reconstruction of the event stream.

Table of Contents

Preface . . . x

1.0 Introduction . . . 1

2.0 Background . . . 4

2.1 Event-Based Vision Sensors . . . 4

2.2 Aperture Robust Multi-Scale Flow . . . 5

2.3 System-on-Chip Development . . . 8

2.4 Event-Stream Compression . . . 9

3.0 Approach . . . 11

3.1 Software Optimization . . . 11

3.1.1 Optimized Algorithm . . . 11

3.1.2 Complexity Analysis . . . 14

3.2 Hardware Acceleration . . . 16

3.2.1 Event Accumulation . . . 18

3.2.2 Window Arbitration . . . 19

3.2.3 Stream Averaging . . . 20

3.2.4 ARMS Compute Core . . . 22

3.3 Flow-Based Stream Compression . . . 23

3.3.1 System Timing . . . 24

3.3.2 Event Transmission . . . 25

3.3.3 Event Prediction . . . 26

3.3.4 Evaluation Metrics . . . 28

3.3.4.1 Compression . . . 28

3.3.4.2 Event-Stream Distance . . . 29

4.0 Experiments and Results . . . 31

4.1 fARMS and hARMS Evaluation . . . 31

4.1.1 Trivial Pattern . . . 31

4.1.1.1 Direction Estimation Accuracy 32
4.1.1.2 Throughput 35
4.1.1.3 Resource Utilization 38
4.1.1.4 Estimated Power 41
4.1.2 hARMS on Real-World Datasets 42
4.1.2.1 Dynamic Rotation Dataset 43
4.1.2.2 MVSEC Dataset 45
4.1.2.3 Pendulum VGA Dataset 47
4.1.3 Performance Comparisons 48
4.1.3.1 Embedded Performance 49
4.1.3.2 Desktop Performance 51
4.2 Event-Stream Compression Evaluation 52
4.2.1 Parameter Characterization 52
4.2.2 Temporal Behavior 54
4.2.3 Real-World Dataset Performance 56
5.0 Conclusion 58
6.0 Future Work 60
Bibliography 61

List of Tables

3.1 Algorithm configuration parameters. 11

4.1 Resource usage per accelerator core for various η implemented on Zynq-7045 SoC. 40

4.2 fARMS and hARMS throughput performance comparison for various dataset scenes on Zynq-7045 embedded platform. Real-time operation indicated in bold. 50

4.3 Original ARMS vs faster ARMS (fARMS) throughput performance comparison for various dataset scenes. Real-time operation shown in bold. 52

4.4 FBC performance on real-world datasets. 56

List of Figures

2.1 The principle of aperture robust optical flow computation from a set of local optical flows estimated from increasingly large spatial regions of interest using the ARMS framework. 6

3.1 High level Zynq-SoC acceleration architecture showing the design of hardware ARMS (hARMS). The processing system (PS) and programmable logic (PL) regions, along with the DDR memory shown are hardware components of Zynq platform. 16

3.2 Conceptual diagram for flow-based compression system. 24

3.3 System timing diagram showing relationship between send time (ST), predict time (PT), and predict interval (PI) in relation to the system state. 25

4.1 Bar-Square results are shown for the two directions of motion, up (top) and down (bottom). Local-flow results are shown using red vectors and hARMS output is show in blue. 33

4.2 Flow direction estimates standard deviation for different design configurations. The value of W_m is constant at 320 for all results shown and N is fixed at 1000. 34

4.3 hARMS standard deviation results for different values of η as N changes. Each point represents the average standard deviation across all values of P tested at that configuration. 35

4.4 Maximum throughput results using qVGA Bar-Square dataset. Hardware configurations are denoted as hARMS-P. 36

4.5 Maximum speedup results using Bar-Square dataset. The speedup is measured over the optimized fARMS design outlined in Section 3.1. W_m is equal to 320 in all cases and N is 1000 for all cases. 37

4.6 hARMS throughput measured using Bar-Square dataset for varying buffer lengths and numbers of windows. P is fixed at 1 for all cases. 38

4.7 LUT utilization vs P for different number of spatial windows. Utilization is represented as a percentage of the 218600 total LUT available on the Zynq-7045 SoC used. 39
4.8 FF utilization vs P for different number of spatial windows. Utilization is represented as a percentage of the 437200 total FF available on the Zynq-7045 SoC used. 40
4.9 BRAM utilization vs N for different number of spatial windows. Utilization is represented as a percentage of the 545 total 36Kb BRAM blocks available on the Zynq-7045 SoC used. 41
4.10 Estimated dynamic FPGA power consumption vs P for different values of η. Estimates are obtained from Xilinx Vivado after bitstream generation is completed. Results are for implementation on the Zynq-7045 SoC and do not include the power consumption of the ARM processing system. N is 1000 for all estimates. 42
4.11 Qualitative results showing correction of local-flow estimates using the hARMS design. The panels show local and hARMS flow direction estimates for events recorded using DAVIS. The events are overlaid on grayscale images captured using the DAVIS. 44
4.12 Comparison of hARMS results with IMU ground truth for dynamic rotation DAVIS dataset. 45
4.13 Comparison of hARMS flow results with the provided ground truth for four MVSEC scenes. 46
4.14 hARMS results for crossing pendulums at different visual depths. The event frames shown are accumulated over 20 ms of motion and only the relevant portion of the VGA sensor frame is displayed. 48
4.15 Characterization of predict time's impact on evaluation metrics. 53
4.16 Characterization of predict interval's impact on distance and total number of events in the reconstructed stream (predicted events). 54
4.17 Reconstruction performance over time compared ER achieved with random removal of events. 55

Preface

I dedicate this thesis to my parents. I am grateful for their support, which has enabled my academic pursuits, including this research.

This research was supported by the NSF Center for Space, High-performance, and Resilient Computing (SHREC) industry and agency members and by the IUCRC Program of the National Science Foundation under Grant No. CNS-1738783. I thank the committee members—Dr. Alan George, Dr. Ryad Benosman, and Dr. Rajkumar Kubendran—for their time and support of me throughout this research. I especially thank Dr. Himanshu Akolkar for his support of this research through countless invaluable technical discussions.

I also thank all of my fellow SHREC students for their support and input throughout the course of this research, especially the students of P1. Finally, I thank my friends and family, especially my fiancé, Lauren, for their support and patience throughout this research.

1.0 Introduction

The emergence of event-based vision sensors has led to the development of new apps and algorithms that are able to leverage the high temporal resolution that these sensors provide. One such application is the computation of optical flow. Traditional optical flow algorithms such as the Horn and Schunk [15] and Lucas-Kanade [22] methods operate on traditional camera frames and thus are not suitable to take advantage of the high temporal resolution and asynchronous characteristics of event-based vision sensors. Many new methods for computing optical flow from event-based sensors have been developed to capitalize on the unique characteristics of these sensors. One such method is a modified version of the Lucas-Kanade method that uses the events to determine spatial and temporal gradients [5]. Another method, uses the derivative of the regularized surface of events to estimate the magnitude and direction of an object's motion [4]. Both methods discussed are susceptible to the aperture problem of optical flow, i.e, they will produce flow vectors that are normal to the moving edge regardless of the true direction of motion. This is due to the fact that both methods view only a local region of events and do not consider all the events produced by an object moving through the scene.

There are some methods of event-based optical flow calculation that have been able to address the aperture problem and compute the true optical flow using event data. One such method, known as EV-FlowNet, is presented in [28]. This method uses a self-supervised neural network to generate the optical flow using frame based accumulation of events. This method accumulates events over a certain time window before calculating the flow for the whole frame. It is also dependent on the use of quality grayscale images generated by the DAVIS event camera for training, making it less adaptable to varied and unpredictable visual scenes [28]. Akolkar et. al. [1] proposed an unsupervised event-per-event spatial pooling of local-flow computations to solve the aperture problem while calculating flow asynchronously using only generated temporal contrast events. They present a method referred to as aperture robust multi-scale (ARMS) for computing optical flow from an event stream [1]. Details of the ARMS algorithm are discussed further in Section 2.2.

In this research, we propose a redesigned and optimized ARMS algorithm referred to as faster ARMS (fARMS), and analyze time complexity when compared to the original ARMS algorithm. We then present a hardware acceleration architecture of the fARMS flow algorithm using a hybrid system-on-chip (SoC) embedded platform containing a field programmable gate array (FPGA). We will refer to this architecture as hardware ARMS (hARMS). fARMS and hARMS were developed to complement the asynchronous nature of event-based vision sensors and the events they output. The hARMS architecture allows for flexible configuration based on application specific needs and provides significant improvements in latency and throughput compared to both the existing ARMS flow algorithm and the fARMS software baseline. This improved performance allows for real-time operation in a variety of visual scenarios, opening up the possibility of more widespread optical-flow-based app deployment on embedded platforms.

Although there are a variety of event-based optical flow techniques, relatively little research has been done on hardware acceleration of them. This is due in part to the fact that some of the local-flow algorithms are able to perform in real time without hardware acceleration because they only consider a small amount of local data. There has, however, been some related research in this area.

A block-matching optical flow algorithm for event-based sensors was implemented on an FPGA by Liu and Delbruck [20]. This implementation, however, was found to perform poorly in real-world scenes and was therefore expanded on in [21] to improve performance. It is estimated in [21] that the improved design would require 100k look-up tables (LUTs) and 35k flip-flops (FFs) when implemented on an FPGA using a similar architecture as proposed in [20]. Block matching was shown, in some cases, to find the true optical flow, however this was dependent on a predetermined block size parameter and the dynamics of the scene [21]. Although this method can overcome the aperture problem in some cases, it still fails to fully make use of the high temporal resolution of event-based cameras. The algorithm is not asynchronous, but rather operates on time slices of accumulated events, therefore sacrificing temporal resolution.

Another example of event-based optical flow acceleration using an FPGA is presented in [2]. This research presents an FPGA implementation for a modified version of the iter-

ative derivative of the surface of events algorithm (sometimes referred to as plane-fitting) presented in [4]. The algorithm derives the surface of events after a temporal regularization to asynchronously estimate the flow, and is capable of performing at a throughput of 2.75 Mevt/s. However, the overall throughput of the system is limited to 1.46 Mevt/s due to the pre-processing stage [2]. The design was implemented using a Xilinx Spartan 6 LX150 FPGA on an Opal Kelly XEM6010 board and required 3794 logic slices, 138 block RAMs (BRAM), and 16 DSPs when using a 304×240 *px* resolution ATIS sensor.

While results presented in [2] show impressive throughput, allowing for real-time, asynchronous operation, it makes no attempt to address the aperture problem. The design presented also has high BRAM requirements because recent events at each pixel location are stored. This indicates that the architecture would scale poorly as the resolution of the event-based vision sensor increases, drastically increasing BRAM requirements and potentially exceeding the available resources on many embedded FPGA platforms. Finally, an implementation of the surface of events approach [4] using spiking neural networks and a neuromorphic spike based processor can be found in [14].

After presenting the fARMS and hARMS event-based optical flow algorithms, they are applied to a novel flow-based event stream compression method. Although compression for asynchronous event streams has been demonstrated in the literature [3, 6, 10, 12, 16], no prior work has leveraged the added information provided by optical flow to achieve compression of event streams. The flow-based compression (FBC) presented in this research can be coupled with existing event stream compression methods to achieve high compression ratios while maintaining the quality of the decompressed event stream. This will enable communication of event data on systems with strict power and bandwidth constraints, without significantly impacting the performance the apps receiving the event stream. Background on existing methods for event stream compression are discussed in Section 2.4.

2.0 Background

This section provides an overview of event-based vision sensors and their principle of operation. The ARMS flow algorithm used as the basis of this research is discussed in further detail. A brief discussion of the SoC design methodology and tools used in this research is provided. Finally, a discussion of event-stream compression is provided with background on the relevant literature.

2.1 Event-Based Vision Sensors

Unlike traditional cameras, which sample pixel intensity at a fixed, synchronized frame rate, event-based vision sensors record asynchronous pixel events. These events encode temporal log intensity contrast at a pixel as either an "on" or "off" event for increasing and decreasing intensity over time respectively [13]. The sensor outputs events using address-event representation (AER), where each event packet includes the x and y coordinates of the event pixel, the event time t, and the event polarity p [8].

The operating paradigm of event-based vision sensors provides multiple advantages over traditional cameras such as high dynamic range and high temporal resolution [13]. High dynamic range allows for detection and tracking of objects in extreme lighting conditions where traditional cameras would be saturated and unable to detect objects. The high temporal resolution of event-based vision sensors is of particular interest for the optical flow application. The microsecond precision of the sensors allows for the possibility of highly accurate optical flow estimates even when objects are moving rapidly through the scene.

This research uses data recorded with a variety of event-based vision sensors with resolutions ranging from 240×180 px to 640×480 px [9, 24, 26]. Event cameras often provide lower resolution than traditional cameras, with early versions having resolutions lower than the ones used in this research [19]. However, as the technology has matured, higher resolution sensors such as the 1280×720 px sensor presented in [11] have been developed. Some

event-based vision sensors are also able to record grayscale intensities either as synchronous images or asynchronous events along with the temporal contrast event outputs. The calculation of optical flow using ARMS, however, only requires the use of temporal contrast events, therefore grayscale data is not considered.

2.2 Aperture Robust Multi-Scale Flow

Optical flow computations in general use the movement of pixel-based information such as intensity or events to calculate direction of motion. These true direction estimates, however, are hard to estimate due to the "aperture problem" which arises when the flow computation is performed only on a section of the object. To reliably compute the true direction of motion the flow algorithm needs information about the motion of the whole object. This requirement, however, can lead to a computational bottleneck as it requires additional steps such as segmentation or clustering of the different objects in the scene. Recently, a novel method, called ARMS was proposed [1], which overcomes these problems in an event-by-event, unsupervised manner that eliminates the need for additional computations. A description of the basic principles behind the working of this method are provided in this section.

Fig. 2.1 shows the operational principle of the ARMS method. Consider a contour shown in gray moving in a horizontal direction indicated by the blue arrow with the true-flow velocity of U. For each event generated by this moving contour, the local flow (U_n) can be computed using the derivative of the surface of events. The direction of this local flow is always orthogonal to the local tangent of the contour and typically does not reflect the true direction of motion. However, the true flow and local flow are related based on the orientation of this edge and the true-flow direction as shown in (2-1). Interestingly, as the orientation of edge at which local flow is computed becomes orthogonal to the true-flow direction, the local-flow magnitude goes to its maximum value and is equal to the true-flow magnitude, as the value of $cos(\theta)$ goes to one.

$$\mathbf{U_n} = |\mathbf{U}| \cos(\theta) \tag{2-1}$$

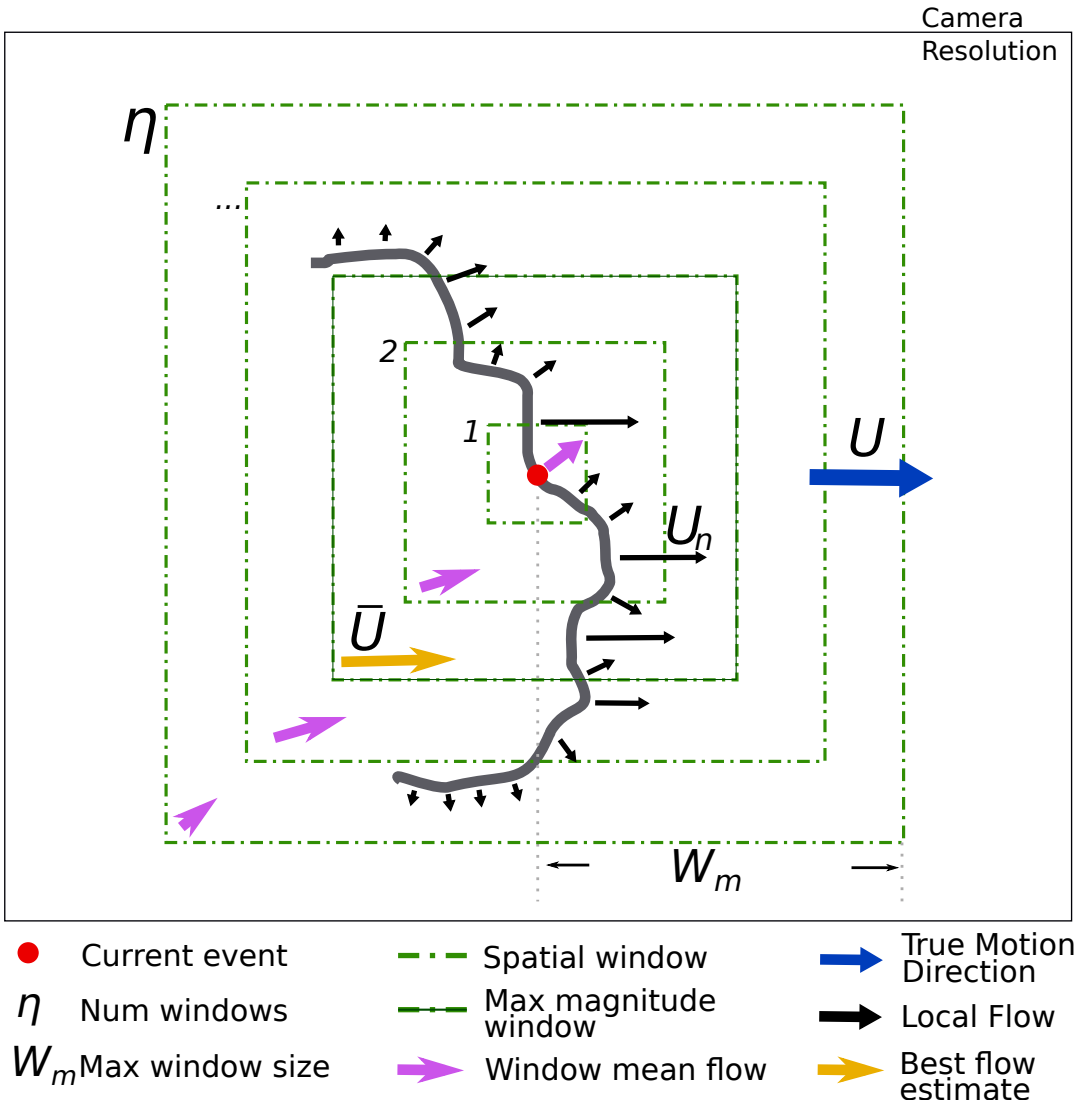

Figure 2.1: The principle of aperture robust optical flow computation from a set of local optical flows estimated from increasingly large spatial regions of interest using the ARMS framework.

In ARMS, this principle is used to show that it is possible to find a spatial neighborhood window around an event, in which maximizing the average local-flow magnitude is equivalent to minimizing the error difference between average local-flow direction and the true-flow direction. This operation can be summarized in (2-2) where k is the different spatial neighborhood sizes. This means that the minimization problem in (2-2) can be used to search for

the best spatial neighborhood size. This neighborhood size corresponds to the best "aperture size" for the flow computation based on the different objects present in the scene, which is determined without requiring any a-priori knowledge of the scene itself, thus, operating in a fully unsupervised manner while performing the flow computation event-by-event.

$$\arg\min_{k}(E) = \arg\min_{k}(|\mathbf{U}| - \overline{|\mathbf{U}_n|}) \equiv \arg\max_{k}(\overline{|\mathbf{U}_n|}) \tag{2-2}$$

The true flow estimate from this spatial window is then given as the average of all the recent local-flow vectors computed within this window.

The visualization in Fig. 2.1 shows the ARMS realization of this windowing strategy, known as multi-scale pooling. The event at which the true flow is being computed is shown as the red dot. The events on the contour around this event also have already computed, local-flow vectors shown as black arrows. The magnitude of these flow vectors (i.e the length of the arrows) varies depending on the contour section orientation w.r.t the true-flow direction. There are η windows of increasing size created (shown as green dashed rectangles), starting from the smallest window centered around the event, up to the largest window of size W_n. For each of these spatial windows the average magnitude ($\overline{|U_n|}$) of all the local-flow vectors within the window (shown as magenta arrows at the bottom left for each window) is computed. Then the window which has the maximum magnitude (shown as green solid rectangle with orange vector in bottom left) is found. The average local-flow vector ($\overline{U}$) in this window is then assigned as the true optical flow for the event.

It may be noted that the theoretical basis for the ARMS flow algorithm holds for any local flow that fulfills the criteria in (2-1). This means that many different existing methods could be used to calculate local flow and the multi-scale search then could be used to correct the flow direction. It is reported in [1] that the ARMS method performed considerably faster than any existing state of the art method, without need for specialized computational hardware such as a GPU. However, for some datasets and scenarios, real-time performance could not be achieved due to high event rates. The performance of the original ARMS algorithm is constrained by two primary factors: 1) The ARMS algorithm requires computation of the average local-flow vectors on all events within each window leading to repetitive averaging as the larger windows already encompass the events of the smaller neighborhoods and 2) the

averaging considers all pixel locations in the η windows even if no new recent events occurred at these pixels. This means that the computational complexity of ARMS depends on both the maximum search window size (W_m) and the number of windows (η). Further analysis of the ARMS algorithm complexity is presented in Section 3.1.2.

To mitigate these issues with the ARMS algorithm, an optimization of the ARMS algorithm is proposed to achieve significantly higher throughput. We refer to this new optimized ARMS algorithm as fARMS. The optimization strategy and the new fARMS algorithm is discussed in detail in Section 3.1.1. Further, we realize a high-performance parallel implementation in hardware on a Xilinx Zynq-7000 series SoC. This hardware implementation of ARMS is referred to as hARMS and its performance improvements, throughput, accuracy and different parameter considerations are detailed in later sections.

2.3 System-on-Chip Development

The Xilinx's Zynq-7000 series SoC is used to implement the hARMS architecture presented in this paper. Xilinx's Zynq SoC is a hybrid computing platform that couples a traditional ARM processing system (PS) with a programmable logic (PL) FPGA fabric region. This architecture allows for algorithms to be split across both the regions of the SoC to achieve optimal performance. To streamline the development of accelerated apps targeting the Zynq platform, Xilinx provides the Software Defined System-on-Chip (SDSoC) development tool. SDSoC uses high-level synthesis (HLS) to allow hardware designs to be written in C/C++ for rapid development and verification of designs. SDSoC and HLS were used in the development of hARMS to allow for efficient design iteration and increase the configurability of the design.

Specifically, the Zynq-7045 SoC was used, which contains a dual-core ARM Cortex-A9 processor operating at 667 MHz and a Kintex-7 FPGA [27]. This device was chosen due to its use in embedded computing applications combined with its large FPGA fabric to allow scaling to large hARMS configurations. hARMS can also be deployed on other Xilinx SoC platforms depending on the desired application, computing environment, and configuration.

2.4 Event-Stream Compression

The operation of event-based vision sensors outlined in Section 2.1 performs inherent compression of the visual information in the scene by only transmitting information when the illuminance at a given pixel changes [9]. While events are spatially sparse due to this inherent compression, the high temporal resolution of event-based vision sensors can result in high event rates in active areas of the scene. These high event rates can become prohibitive for communication, especially for embedded platforms with significant power constraints. To overcome high event rates, more research focus has been placed on the compression of these event streams in recent years. Like traditional image or video compression, event-stream compression can be classified as lossless or lossy depending on whether information is lost between the original event stream and the decompressed event stream. There has been research exploring both types of compression for event-streams.

A lossless compression method using spike coding was introduced in [6] and expanded upon in [10]. This method divides the event stream into multiple event cubes in space and time, which are then individually encoded by capturing the spatial and temporal redundancies in the local cube of the event stream. The cubes are encoded using both adaptive octree-based cube partitioning and intracube prediction [10]. Other lossless methods come from existing, general-purpose compression techniques such as entropy coding, dictionary-based compression, fast integer compression, and compression specific to the Internet of Things (e.g. Sprintz [7]). The application of these methods to event-based vision-sensor data has been explored in [16] and compared to the spike-coding method in [10]. They find that the dictionary-based Lempel-Ziv-Markov chain algorithm provides the highest compression ratio when the sensor is static and that the spike-coding method provides the highest compression ratio when the sensor is in motion. Another method for lossless compression of event streams is known as time-aggregation-based lossless video encoding for neuromorphic vision sensors (TALVEN) [17]. As the name suggests, this method relies on time aggregation of events followed by frame coding. While this method was shown to result in good compression ratios, the time aggregation and frame-based nature make it less appealing in applications with low-compute resources and low-latency requirements. An algorithm that

operates asynchronously on the event data is more desirable due to its potential to achieve lower event delays. These lower event processing delays are enabled by immediate processing of events rather than waiting until the end of a temporal window to process events in the frame. Asynchronous operation can also reduce the memory footprint of the algorithm by operating on a small history of recent events rather than accumulating a large frame of events.

Compared to lossless compression, the use of lossy compression methods for event streams has been explored relatively little. One method for lossy event stream compression is the use of quad trees and Poisson disk sampling as introduced in [3]. This method achieved compression ratios of more than 6× those presented in prior works. Although this work achieves high compression ratios, it is limited by its dependence on the use of intensity images generated by an event-based vision sensor such as the DAVIS or RGB-DAVIS [3]. This limitation is a significant drawback as it cannot be used with purely asynchronous temporal contrast sensors, but instead relies on additional intensity data to select the low-priority regions where events can be removed without impacting the quality of event reconstruction. It is desirable to have a lossy compression technique that can be applied only to events. Such a method is introduced in this research through the use of flow-based prediction.

3.0 Approach

The following sections discuss the approach for the three focuses of this research. Software optimization of the ARMS algorithm for event-based optical flow is introduced. The developed hardware acceleration architecture is then discussed in detail. Finally, a novel method for event-stream compression using optical flow is presented.

3.1 Software Optimization

This section outlines the optimizations made to the ARMS algorithm. The redesigned algorithm, referred to as faster ARMS (fARMS), is presented in detail. Complexity analysis is performed for both the ARMS and fARMS algorithms, and results are compared.

3.1.1 Optimized Algorithm

Table 3.1 introduces the parameters used to configure the different ARMS algorithms. The configuration of these parameters impacts the performance and accuracy of the design and must be set based upon the performance requirements of a given application. W_m, η, and τ are configured for all implementations whereas N is characteristic only of fARMS and hARMS and P is only used for hARMS configuration.

Table 3.1: Algorithm configuration parameters.

Parameter	Description
W_m	maximum window size for multi-scale pooling
η	number of spatial windows around event
τ	refraction time for flow events in ARMS algorithm
N	length of recent flow event buffer
P	number of parallel accelerator cores (hARMS only)

The ARMS algorithm, presented in [1], relies on successive pooling of events in multiple expanding spatial windows. This design results in repetitive computation of averages in regions around the event that are a subset of multiple spatial windows. It also requires searching the whole $(2W_m) \times (2W_m)$ px region around the most recent event, regardless of which, or how many, of the pixels in that region have triggered recent events. This inefficiency is introduced by the reliance on a frame of recent events. The use of an event frame does not align with the asynchronous nature of the event stream generated by the sensor, which has no concept of a frame. We therefore abandon the use of an event frame altogether, and present a more efficient design based on the use of a small time history of recent events.

The redesigned algorithm used for fARMS is presented in Algorithm 1. The first, and most significant, optimization is the introduction of the Recent Flow event Buffer (RFB). This buffer stores the last N events generated with a valid local flow. Because the ARMS algorithm only requires the recent-flow events from the last τ ms, no information will be lost as long as the value of N is greater than or equal to the number of valid events in that time window. In fact, the use of the RFB preserves more information than the use of an event frame. This is because the event frame only preserves the most recent event at each pixel, discarding the older event even if it may have fallen into the τ ms time window. The RFB however, has no limitation on the number of events per pixel that can be stored because the location of the event is explicitly included for each entry instead of being implicitly encoded in the event frame location. We hypothesize that multiple events at a single pixel within the τ ms window are most likely to occur along strong edges in the scene where the local-flow estimates will be most accurate. Because of this, it is expected that, on average, improved true-flow estimates from the fARMS algorithm will be observed when compared to ARMS.

While improved accuracy from fARMS is expected, the primary objective is optimization for improved throughput performance. The use of the RFB also yields significant performance improvement due to the removal of redundant computation and the reduction of the search space. To enable the use of the RFB and achieve this performance improvement, the challenge of determining which windows each event in the RFB falls into needs to be addressed. Unlike the event frame, the RFB maintains no spatial relationship between events, instead just storing the x and y locations of the event. Therefore, we introduce a window ar-

Algorithm 1 fARMS algorithm for true flow.

$RFB[N] \longleftarrow 0$, recent flow-event buffer
$next\_idx \longleftarrow 0$, RFB fill index
$EDGE[\eta + 1]$, window bin edges

1. Initialize Window Edges
for $win \longleftarrow 0$ to η **do**
  $EDGE[win] = win \cdot (W_m/\eta)$
end for

2. Process Events
for each $event(x, y, t, vx, vy, mag)$ **do**
  $sums \longleftarrow 0$, holds vx, vy, and mag sum arrays
  $COUNTS \longleftarrow 0$, window event count array
  $RFB[next\_idx] = event$
  $next\_idx = (++\ next\_idx) \bmod N$
  for $i \longleftarrow 0$ to N **do**
    if $\text{abs}(RFB[i].t - event.t) \leq \tau$ **then**
      2a. Window Arbitration
      $dx = \text{abs}(event.x - RFB[i].x)$
      $dy = \text{abs}(event.y - RFB[i].y)$
      $d_{max} = \max(dx, dy)$
      for $j \longleftarrow 0$ to $\eta - 1$ **do**
        if $d_{max} \in [EDGE[j], EDGE[j+1][$ **then**
          $tag = j$
        end if
      end for
      2b. Window Averaging
      for $k \longleftarrow 0$ to $\eta - 1$ **do**
        if $tag \leq k$ **then**
          $sums.VX[j]$ += $RFB[i].vx$
          $sums.VY[j]$ += $RFB[i].vy$
          $sums.MAG[j]$ += $RFB[i].mag$
          $COUNTS[j]$ ++
        end if
      end for
    end if
  end for
  $MAG\_AVGS = sums.MAG/COUNTS$
  $w_{max} = \text{argmax}(MAG\_AVGS)$
  $true\_vx = sums.VX[w_{max}]/COUNTS[w_{max}]$
  $true\_vy = sums.VY[w_{max}]/COUNTS[w_{max}]$
  return **Flow**$(true\_vx, true\_vy)$
end for

bitration technique to give each event in the RFB a window tag based on its x and y location relative to the current event. First, maximum component distance between the current event and the RFB event is found. Then a tag is assigned based on the pre-computed window bin that the maximum component distance falls into. It is known that an event that falls into a given spatial window will also belong to all larger spatial windows. This means that only $\eta + 1$ unique window tags are needed to encode all possible windows along with the scenario where an event is not included in any windows. Once the windows that an event falls into are determined, the averaging can be performed as shown in part 2b of Algorithm 1. With the use of window arbitration and the RFB, the algorithm only requires iteration over all of the N events in the RFB as opposed to costly searches over each of the expanding spatial windows in a frame of events.

The true-flow results from fARMS are calculated in the same way as ARMS. The spatial window with the maximum local-flow magnitude is considered the correct window and the averages of the x and y components of local flow in that window are returned as the true-flow result. A comparison of the complexity of both ARMS and fARMS is provided in the following section and accuracy results are discussed in Section 4.

3.1.2 Complexity Analysis

The worst-case complexity of both the ARMS and fARMS algorithms is evaluated based on the number of loop iterations required for the true-flow computation for a single event. The number of loop iterations, n_{ARMS}, for the original ARMS algorithm in [1] is shown in (3-1).

$$n_{ARMS} = \sum_{i=1}^{\eta} \left(\frac{2W_m}{\eta} \right)^2 i^2 \tag{3-1}$$

Expansion of the summation in (3-1) yields the expression for n_{ARMS} given in (3-2). From (3-2) the complexity in terms of loop iterations is derived and shown in (3-3).

$$n_{ARMS} = \frac{1}{6} \left(\frac{2W_m}{\eta} \right)^2 \eta \left(\eta + 1 \right) \left(2\eta + 1 \right) \tag{3-2}$$

$$n_{ARMS} \in O(W_m^2 \eta) \tag{3-3}$$

From (3-3) it can be seen that the complexity of the ARMS algorithm is bounded by W_m^2 and η. The squared dependence on W_m poses significant challenges for the scaling of the algorithm, especially as the resolution of the sensor increases. To capture the same region of the scene within the spatial windows, W_m must scale as the sensor resolution scales, otherwise an insufficient portion of the scene may be considered for evaluating the true flow at an event. This could reduce robustness to the aperture problem and the overall effectiveness of the algorithm. Furthermore, it is later observed in Section 4 that flow accuracy tends to improve with larger values of W_m, particularly when there is a single dominant direction of motion in the scene. The necessity of scaling to higher values of W_m in many cases means that complexity that scales independent from W_m, and thus sensor resolution, is highly desirable.

The number of loop iterations required to compute true flow for one event using the fARMS algorithm is shown in (3-4). From this the overall complexity is derived in (3-6).

$$n_{fARMS} = \sum_{i=1}^{N} 2\eta \tag{3-4}$$

$$n_{fARMS} = 2N\eta \tag{3-5}$$

$$n_{fARMS} \in O(N\eta) \tag{3-6}$$

From (3-6) it is seen that the fARMS complexity is bounded only by N and η. Therefore, the fARMS algorithm achieves the objective of scaling independent of W_m. Since both ARMS and fARMS complexity scales with η, we compare W_m^2 and N to evaluate the relative scaling of both algorithms. In most cases W_m^2 is much larger than N, meaning fARMS has a much lower run-time complexity. Take for example a benchmark configuration that will be used in Section 4.1.1 where $W_m = 320$, $\eta = 4$, and $N = 1000$. When substituting these parameters into (3-2) and (3-5) it yields $n_{ARMS} = 768000$ and $n_{fARMS} = 8000$. In this case the fARMS algorithm results in a 98.96% reduction in the theoretical complexity of the true-flow calculation when compared to the original ARMS algorithm. While the complexity

difference will vary as the values of W_m and N are changed, this analysis shows that the fARMS algorithm substantially reduces ARMS computational complexity.

3.2 Hardware Acceleration

The hARMS system architecture describes the hardware realization of the fARMS algorithm for improved performance on embedded platforms. The system architecture was designed to be modular to allow for streamlined realization of various configurations. The result is a flexible acceleration architecture that can be adapted to application-specific needs. Fig. 3.1 shows this system architecture with each of the main processing modules included.

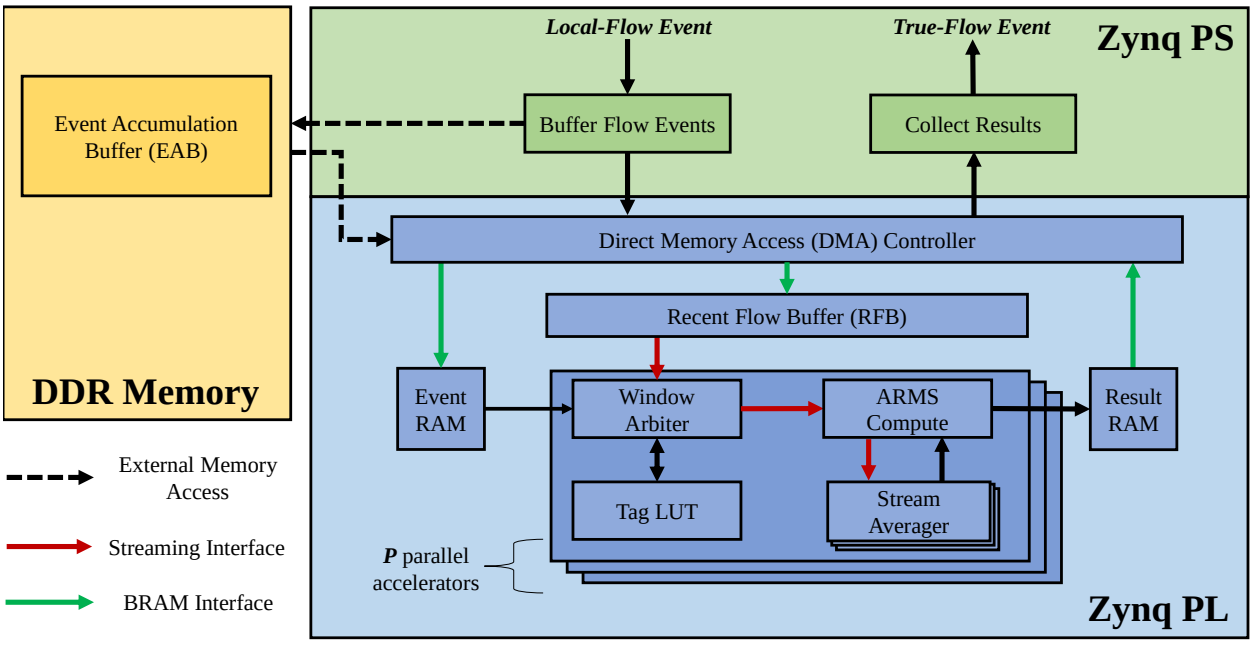

Figure 3.1: High level Zynq-SoC acceleration architecture showing the design of hardware ARMS (hARMS). The processing system (PS) and programmable logic (PL) regions, along with the DDR memory shown are hardware components of Zynq platform.

The design is divided across three sections of the Zynq SoC platform—DDR memory, processing system (PS), and programmable logic (PL). The DDR memory is used to buffer flow information and store software application variables. The PS consists of a dual-core ARM Cortex-A9 processor and is used to run the main C++ application that receives local-flow inputs, calls the hardware accelerator, and collects the true-flow results. The PL is used to realize the hARMS accelerator using custom hardware blocks implemented in the FPGA fabric.

The architecture developed for validation and testing computes the local flow in software on the PS. However, the local flow is only considered as an input to the hARMS design and can therefore be computed using any method, including PL acceleration as required by the application. As events are generated and the local flow is computed, the PS is used to accumulate local-flow events and collect the true-flow results. The accumulated events for which a valid local flow exists, and thus true flow can be calculated, are transferred to temporary RAM in the PL fabric using a direct memory access (DMA) controller and a block RAM (BRAM) interface. In addition to being stored in temporary RAM, the accumulated events are also added to the RFB to be used in the processing of future events. The true flow at each accumulated event is then processed in parallel as the RFB is streamed through the P parallel accelerators in the PL region. The processing is performed in a hierarchical order by the window arbitration, stream averaging, and ARMS compute modules. The design of these modules is discussed further in the following sections.

The hARMS design aims to closely match the event-by-event results of the fARMS algorithm. However, for optimization of the hardware implementation, some quantization from full floating point representation is performed. The local-flow results are rounded and represented as 16-bit integer inputs, while the resulting true flow is represented as a 32-bit fixed point value with eight fractional bits. Arbitrary bit width representations are used internally in the design to achieve more resource efficient hardware. Further reduction in the number of bits used to represent data could be considered in some cases depending on the expected range of velocities in the scene or tolerable loss in flow accuracy. However, the standard hARMS configuration presented was designed with the goal of being robust to wide variations in scene dynamics, while maintaining accuracy equivalent to fARMS.

The architecture includes all the configurable parameters available in the fARMS algorithm, with the addition of P as outlined by Table 3.1. Unlike the fARMS algorithm in software, which can define these parameters at runtime, the hardware nature of the hARMS architecture requires configuration before compilation. The value of P is used to specify the number of parallel accelerators to be used in the PL fabric. The high levels of parallelism that can be achieved through the use of this parameter have substantial impact on the design's performance and resource utilization. This impact along with the trade-offs between other parameters will be discussed and analyzed in Section 4.

3.2.1 Event Accumulation

Event accumulation is the primary function of the PS in the system architecture. As local-flow events are generated, the flow components and flow magnitude are stored in the Event Accumulation Buffer (EAB). The EAB holds the events for which the true flow will be calculated when the hardware accelerated function is called. The depth of the EAB is equal to the number of parallel accelerators, P, included in the design. When the EAB is filled with new events, a DMA transaction is initiated to move the EAB data to the PL region via a BRAM interface. This transaction begins the true-flow calculation process for the events in the EAB. Once the EAB is transferred to the PL, the events are added to the RFB as well as to temporary RAM. The RFB is a BRAM ring buffer of length N that is implemented in the PL fabric and retains its values between calls to the hardware accelerator. The N parameter dictates the number of events stored in the RFB and processed in the PL. The ring buffer allows for new events to be added while replacing the oldest events at the same time. This way the most recent N events are always available for each true-flow calculation. When the true-flow calculations begin, the data stored in the RFB is streamed into the accelerator modules as shown in Fig. 3.1. Since the hARMS accelerators are designed for streaming inputs, one value from the RFB can be read each clock cycle, achieving an initiation interval of one and preventing the need for resource-expensive array partitioning.

The hARMS accelerators are designed to perform asynchronous computation, such that, as the ARMS true flow is being calculated using the FPGA accelerator, the EAB can begin

to be filled asynchronously as the previous EAB events are being processed. Buffering events allows multiple true-flow events to be processed simultaneously using the same RFB. This reduces the BRAM required for simultaneous processing of events, while increasing the throughput of the system. Processing multiple true-flow events in this way results in up to $P-1$ future events being considered for a given event when multi-scale pooling is performed. Because P is typically much smaller than N, this artifact of buffering events has no significant impact on the accuracy of the flow estimate as shown by the results in Section 4.1.1.1.

3.2.2 Window Arbitration

Multi-scale pooling as introduced in Section 2.2 requires averaging over expanding spatial windows around the pixel location of the incoming event at which true-flow is being calculated. The window or windows that a recent flow event will fall into is dependent on the location of the recent event around which multi-scale pooling is being applied. The hARMS architecture uses the same window arbitration technique introduced by fARMS in Algorithm 1. The maximum component distance calculated is used as the input to a hardware lookup table ($tagLUT$), which determines the appropriate window tag. The maximum component distance is equal to $\max(F_{width}, F_{height}) - 1$, where the sensor resolution is $F_{width} \times F_{height}\ px$. This value could be considered the theoretical maximum value of W_m, because it is applied outward from the event in all four directions. However, in reality the value of W_m is not restricted due to the resolution indifferent design of the hARMS architecture.

As discussed in Section 3.1.1, there are $\eta+1$ possible window tags that could be assigned to any recent event as it is streamed into the accelerator, with the additional tag representing a recent event that does not belong to any of the defined windows. The tag can then be represented using only $\lceil \log_2(\eta+1) \rceil$ bits in hardware. These tag bits are appended to the input stream, while the x and y coordinate data is removed from the stream, as all required location information is now encoded in the window tag.

The window arbiter hardware is fully pipelined to allow one new recent-flow event to be read from the internal input stream generated from the RFB on each cycle. The window edge values are statically declared in the $tagLUT$ module and the window search is fully unrolled. This implementation allows the $tagLUT$ module to achieve an interval and latency of one cycle. This high-performance window arbitration is achieved regardless of the relative order in which events are streamed in. This allows for the simple ring buffer realization of the RFB, significantly reducing the control logic for the RFB and eliminating the need for software-based window arbitration before streaming data to the accelerator.

3.2.3 Stream Averaging

Averaging is the fundamental operation of multi-scale pooling. It involves averaging the values of recent local-flow events within each window. This operation is performed by the stream averaging module implemented in the PL fabric. The stream averager is a modified stream-based FPGA implementation of the averaging performed in the fARMS algorithm. It differs in its streaming nature and modular design, which allows parallelism between multiple instances of the module. As events from the RFB are streamed into the module, they are added to an internal array of window sums, where there is one sum for each of the η windows used in the design. This addition, however, is only performed for the sums of the windows that the event falls into. The tag assigned by the window arbitration module is used to select the window sums to which the event value should be added. These steps are outlined further in Algorithm 2.

Algorithm 2 AVERAGER streaming algorithm design.

1: $ISTREAM$ internal input stream
2: $WIN\_SUMS[\eta]$, array of intermediate sums
3: $WIN\_COUNT[\eta]$, count of events in each window
4: $AVERAGES[\eta]$, array of resulting averages
5: $sEvent$, stream input event holding $(tag, value, valid)$
6: **1. Compute Window Sums**
7: **for all** events in $ISTREAM$ **do**
8: $sEvent \longleftarrow ISTREAM$ {load event from stream}
9: **for** $idx \longleftarrow 0$ to $\eta - 1$ **do**
10: **if** $sEvent_{tag} \leq idx$ and $sEvent.valid$ **then**
11: $WIN\_SUMS[idx] \mathrel{+}= sEvent.value$
12: $WIN\_COUNT[idx] \mathrel{+}= 1$
13: **end if**
14: **end for**
15: **end for**
16: **2. Compute Averages**
17: **for** $i \longleftarrow 0$ to $\eta - 1$ **do**
18: $AVERAGES[i] = \dfrac{WIN\_SUMS[i]}{WIN\_COUNT[i]}$
19: **end for**
20: **return** $AVERAGES$

As each recent event is added to the appropriate window sums, a count of events that fall into each window is kept. When the entire length of the RFB has been streamed through the averager, the averages for each of the spatial windows are generated by dividing the sum array by the count of events that belonged to the corresponding window. This division occurs once per window for each true-flow event. No checks for division by zero are required because we are guaranteed to have at least one event—the event for which true flow is being calculated—in each window. Because implementation of many dividers is resource intensive, a limit of four hardware dividers per averager module is enforced. These dividers are reused when the number of windows, and therefore divisions, is increased beyond four. Because this operation only occurs once at the end of the processing pipeline for this stage the added latency of pipelined division instead of fully unrolled division only has a limited impact on the overall latency of the stream averaging stage of processing. The constraint of four could be modified to fit specific application needs based on available device resources.

For efficient implementation of Algorithm 2 in hardware, the loops on lines 9 and 17 are fully unrolled. To facilitate parallel access to the sum and count arrays, they are fully partitioned such that all elements can be accessed and modified concurrently. While the directive

is given to fully unroll the division loop on line 17, the unroll factor will, in implementation, be limited by the limit placed on the number of dividers to be instantiated. The compute window sums loop that reads in each of the recent-flow events from the input stream is fully pipelined with an initiation interval of one such that it can read a new event from the input stream once every clock cycle.

3.2.4 ARMS Compute Core

The ARMS compute core is the main functional control block of each hARMS accelerator instantiated. It receives the tagged event stream from the window arbiter. Using the event timestamps included in that stream and the defined value of τ it performs temporal filtering of the event streams. Any event that occurs more than τ microseconds before the EAB event under consideration is flagged and not considered when performing multi-scale pooling using the stream averaging blocks.

Algorithm 3 ARMS compute core algorithm for hARMS architecture.

1: $event$, EAB event of interest (x, y, t)
2: $sEvent$, stream input event (tag, vx, vy, mag, t)
3: $ISTREAM$, output stream from WINDOW_ARBITER
4: **1. Process Input Stream**
5: **for all** events in $ISTREAM$ **do**
6: $sEvent \longleftarrow ISTREAM$ {load event from stream}
7: **if** abs$(sEvent.t - event.t) \leq \tau$ **then**
8: $valid =$ **true**
9: **else**
10: $valid =$ **false**
11: **end if**
12: $VX\_STREAM \longleftarrow (sEvent.(vx, tag), valid)$
13: $VY\_STREAM \longleftarrow (sEvent.(vy, tag), valid)$
14: $MAG\_STREAM \longleftarrow (sEvent.(mag, tag), valid)$
15: **end for**
16: **2. Stream Averaging**
17: $VX\_AVGS =$ AVERAGER(VX_STREAM)
18: $VY\_AVGS =$ AVERAGER$(VY_STREAM$
19: $MAG\_AVGS =$ AVERAGER(MAG_STREAM)
20: **3. True-Flow Selection**
21: $w_{max} =$ argmax(MAG_AVGS)
22: **return** $(VX\_AVGS[w_{max}], VY\_AVGS[w_{max}])$

The ARMS compute core extracts the three values from the stream that must be averaged: the x component of local flow, y component of local flow, and magnitude of local flow. These values and the window tag are passed to three instances of the stream averaging module in parallel streams. The multi-scale average arrays generated are then collected by the compute core and a maximum search is performed on the flow magnitude average results to find the spatial window with the largest local-flow magnitude. The average x and y components of the local flow in that spatial window are then returned as the true-flow results for the EAB event being processed.

The ARMS compute module functionality described is realized as shown in Algorithm 3. The control flow is modified from that of fARMS to efficiently use streaming interfaces, as well as to capitalize on available parallelism. The process input stream loop is fully pipelined such that one event is read from and written to the window arbiter and averager modules respectively on each clock cycle. The calls to the stream averaging modules in lines 17 to 19 are performed in parallel using task-level, dataflow pipelining. Once the stream averaging is completed, the window index, w_{max}, corresponding to the maximum local-flow magnitude is found and the true-flow results are returned and stored in temporary RAM before the DMA transfer of the results back to the PS for collection.

3.3 Flow-Based Stream Compression

There are many apps that can benefit from the fARMS and hARMS fast event-based true flow methodologies presented. One such app is Flow-Based stream Compression (FBC). The ability to quickly compute accurate flow for events enables the possibility of predicting future events. Thus, events only need to be updated periodically or under certain circumstances where flow cannot be accurately estimated.

The structure of the FBC system introduced is composed of two primary components—the *transmitter* and the *receiver*—as shown in Fig. 3.2. The transmitter is located on the edge with the event-based vision sensor, while the receiver may be another edge device or a more powerful computing platform. Therefore, the objective is to minimize the transmitter-side

computation without degrading compression performance to allow deployment on devices with extremely limited computing resources. As seen in Fig. 3.2, events are passed from the sensor to the transmitter where flow is calculated and the events to send are selected. The selected events and their flow are then transmitted to the receiver where event prediction and stream reconstruction (decompression) occurs. Finally, the reconstructed stream is asynchronously streamed into any app designed for processing event streams. This system enables less data to be transmitted via the remote connection between the transmitter and the receiver, saving power and bandwidth. In the following sections the methodologies for system timing, event transmission, and event prediction and stream reconstruction are presented.

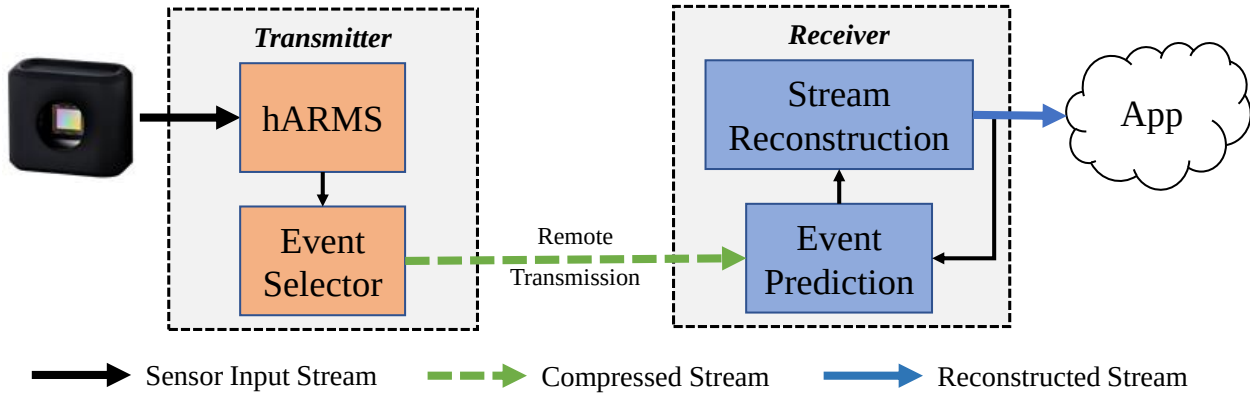

Figure 3.2: Conceptual diagram for flow-based compression system.

3.3.1 System Timing

Before describing the operation of the transmitter and receiver in detail, the system timing is discussed. Synchronization of the transmitter and receiver is performed using the events themselves, preventing the need for any control or handshake messages to be transmitted. Because events already include timestamps and the transmitter and receiver are both configured with the same timing parameters, explicit synchronization is not necessary.

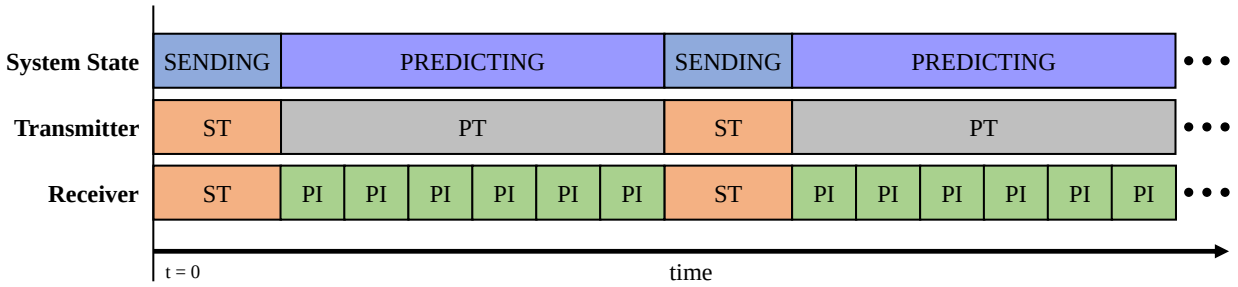

Figure 3.3: System timing diagram showing relationship between send time (ST), predict time (PT), and predict interval (PI) in relation to the system state.

Fig. 3.3 illustrates the synchronization and timing of the system. The system has two distinct states in which it can operate: *sending* and *predicting*. In the *sending* state, all events generated by the sensor are being transmitted with their flow results. During the *predicting* state, the receiver is performing predictions and only events that don't have a valid flow estimate are sent by the transmitter.

There are three modifiable parameters that control the timing of the system. They are the send time (ST), predict time (PT), and predict interval (PI). The send time is the duration that the system is in the *sending* state. The predict time is the duration that the system is in the *predicting* state. Finally, the predict interval determines the time between consecutive predictions of new events by the receiver while the system is in the *predicting* state. An example of the interaction of these parameters and their relationship to the system state are shown in Fig. 3.3.

3.3.2 Event Transmission

When an event is generated by the event camera it is passed directly to the connected embedded platform on which the transmitter is running. When the transmitter receives the event from the sensor it computes the flow of the event using either the fARMS or hARMS architectures presented. The event is then transmitted to the receiver if one of two conditions

is met in the event selector stage: 1) The system is in the *sending* state or 2) the system is in the *predicting* state and the flow cannot be accurately found. For case 1, if flow exists it is sent along with the event. In cases when flow does not exist it is typically due to the local flow quality thresholds (e.g. required inlier percentage for plane-fitting) not being met.

Transmission of an event when no flow estimate is found is required to ensure accurate reconstruction regardless of the system state. This requirement is due to the assumption that events without flow capture new information that would not be predicted based on past flow results. Conversely, it is assumed that events that have a flow during the *predicting* state can be successfully predicted by the receiver using previous flow results and thus do not need to be sent.

3.3.3 Event Prediction

When the system is in the *predicting* state the transmitter is only sending events for which it cannot accurately estimate flow and the receiver is performing periodic predictions based on the defined PI. These predictions aim to reconstruct the event stream using the optical flow estimates received during the previous *sending* state. This task presents some challenges that must be considered in order to effectively reconstruct the event stream.

Algorithm 4 outlines the primary control flow and operations performed for event prediction by the receiver. As discussed earlier, synchronization is based on the events arriving, therefore the interval may not be exactly the same as the defined predict interval. However, this variability is accounted for in the calculation of dt. Additionally, due to noise in the sensor, events consistently arrive during the predicting state, allowing synchronization solely using the event stream. Another challenge during event prediction is sub-pixel motion. The calculated x and y change based on the optical flow will often contain partial pixels. To resolve this, residuals $resX$ and $resY$ are stored and added to the next predicted change for that event.

Algorithm 4 Receiver event prediction algorithm.

```
1: eventStream, reconstructed output stream
2: frame, width×height internal frame of events spatially storing events
3: lastPredictTime, last time a prediction for all events was made
4: PI, predict interval defined by user configuration
5: for each event do
6:   if SENDING then
7:     add event to eventStream
8:   else if PREDICTING then
9:     add event to eventStream
10:    if PI since last prediction then
11:      dt = event.t − lastPredictTime
12:      for all active events (evt) in frame do
13:        1. Compute predicted event motion
14:        dx = evt.vx · dt + evt.resX
15:        dy = evt.vy · dt + evt.resY
16:
17:        2. Create new predicted event
18:        newEvent.t = lastPredictTime + PI
19:        newEvent.x = evt.x + round(dx)
20:        newEvent.y = evt.y + round(dy)
21:
22:        3. Add predicted event to output stream
23:        add newEvent to eventStream
24:
25:        4. Compute new residuals and add predicted event to next frame
26:        newEvent.resX = dx − round(dx)
27:        newEvent.resY = dy − round(dy)
28:        frame[newEvent.x][newEvent.y] = newEvent
29:      end for
30:      lastPredictTime += PI
31:    end if
32:  end if
33: end for
34: return eventStream
```

This method for stream reconstruction using event prediction allows all the work of prediction to be performed by the receiver, which is expected to be a more capable computing platform than the transmitter in most cases. One downside to this method is an observed phenomenon we refer to as 'vanishing events'. Because of the potential for error in the flow estimates, it is possible that the predicted paths of two events will incorrectly cross and be combined. This results in a reduction in the overall number of active prediction events as more predictions are made. The impact of vanishing events can be mitigated by avoiding

overly long predict times and ensuring the send time is selected correctly based on anticipated scene activity. As shown in Section 4.2, while vanishing events and flow errors do have an impact on reconstruction quality, the method is still shown to be capable of producing a reconstructed stream that closely matches the original event stream. The similarity of the reconstructed and original streams are quantified using the metrics outlined in Section 3.3.4.

3.3.4 Evaluation Metrics

To evaluate the performance of the FBC method introduced we consider two key metric areas: compression and event-stream distance. These metrics are discussed in the following sections. The results reported in Section 4.2 use the metrics outlined here.

3.3.4.1 Compression

The first metric considered is the event reduction (ER). This is defined as the fraction of events in the original stream that were never sent from the transmitter to the receiver. The equation for ER is formalized in (3-7), where N_s and N_{tx} are the number of events in the input stream and the number of events transmitted, respectively.

$$ER = \frac{N_s - N_{tx}}{N_s} \tag{3-7}$$

The second metric considered is the compression ratio (CR). This is a commonly used metric in other event-stream compression work such as [16]. CR is defined as the number of bytes in the original data stream divided by the size of the compressed stream in bytes. For this metric we must consider the additional data required for each event that includes flow information. The equation for calculation of CR is shown in (3-8).

$$CR = \frac{N_s \times 8}{(N_{tx} - N_{nf}) \times 11 + N_{nf} \times 8} \tag{3-8}$$

In (3-8) and moving forward it is assumed that flow data is represented as two 12-bit values. This representation requires an additional three bytes to be transferred for each event containing flow. Therefore, flow events are 11-bytes in total while normal AER events are 8-bytes as used in [16, 3]. The value of N_{nf} equals the number of events in the event stream

for which no flow can be calculated. All of these events must be transmitted regardless of the system state, but because they have no flow they do not require the extra bytes of flow information. The difference between the total number of events transmitted and the no-flow events is that events with flow require an additional 3-bytes of information. In this research, no additional compression is applied to the event stream beyond the FBC method presented. In practice, the FBC-compressed stream could be further compressed by any of the existing methods discussed in Section 2.4.

3.3.4.2 Event-Stream Distance

Because the FBC method introduced is a form of lossy compression, evaluation of the performance in terms of event-stream reconstruction accuracy, referred to as event-stream distance, is important. There are various methods presented in the literature that evaluate the similarity of two event streams. Some accumulate event frames and use standard image similarity metrics such as the structural similarity index measure (SSIM) and peak signal-to-noise ratio (PSNR). These metrics are used in [3], however, they acknowledge that this method gives little insight into temporal errors in the reconstructed event stream. They therefore create a new metric to quantify the time error. This metric however, requires the original and reconstructed streams to have the same number of events—an assumption that does not hold for FBC due to vanishing events.

It is desirable to have a metric that captures both spatial and temporal changes and can be applied to event-streams of different lengths. A metric that meets these requirements is the asynchronous spatiotemporal spike metric (ASTSM) proposed in [18], which is used in this research. This method applies a spatiotemporal Gaussian to the event streams to convert them into a reproducing kernel Hilbert space (RKHS). By computing the inner product of the event streams in the RKHS the distance between the two streams can be found. For a more detailed discussion on this method please refer to [18].

To implement this method, we utilize the open-source code provided by the authors of [18]. The algorithm takes the parameters σ_x, σ_y and σ_t for constructing the Gaussian kernel. For computational tractability, the event streams are split spatially and temporally into

"event cubes" between which the distances are calculated. These cubes have parameters W, H, and L for width, height, and temporal length. For all metrics reported in this research the configuration ($\sigma_x = 5\ px$, $\sigma_y = 5\ px$, $\sigma_t = 5000\ \mu s$) is used as used in [18]. For the event cubes the spatial size is set equal to that of the sensor and the value of L to the prediction interval. The resulting distance metric is a spatiotemporal distance where a distance of zero represents perfect reconstruction while increasing distance represents increasing dissimilarity between the original stream and the reconstructed stream. For interpretability of the results in this research the distance computed using the ASTSM method is normalized for each pair of event cubes by the number of events in the original stream that fall into that cube. Distance is therefore reported on a per-original-event basis.

4.0 Experiments and Results

The performance of the fARMS and hARMS optical flow algorithms presented is evaluated in the following sections. Design space exploration is performed to characterize various hARMS configurations. Finally, the results of the FBC method introduced are presented.

4.1 fARMS and hARMS Evaluation

The presented fARMS and hARMS designs are evaluated against the same datasets presented in [1] for a direct comparison of accuracy and performance between the hardware and software designs. These datasets span a variety of scenes and sensor resolutions, allowing for a detailed investigation of the design space. We also evaluate how resource utilization and performance change when different configurable parameters are modified, and analyze design trade-offs when selecting a hardware configuration. A real-time performance comparison for the different datasets used is also provided to show successful achievement of real-time operation across a variety of visual scenes, event rates, and sensor resolutions.

The Xilinx ZC706 development board, which includes the Zynq-7045 SoC, was used for all embedded software and hardware benchmarks. The Zynq's FPGA contains 218k LUTs, 437k FFs, 900 DSP slices, and 19.2 Mb of BRAM. Resource utilization will be considered as a percentage of these total resources. Software benchmarks were run on the Zynq's ARM processor using a single core operating at 667 MHz. Compiler optimization was set to -O3 and Xilinx's PetaLinux distribution was used. All hardware configurations used a 200 MHz clock for both the accelerator and the DMA controller.

4.1.1 Trivial Pattern

The same trivial pattern dataset presented in [1] was used for evaluation of the developed hARMS accelerator. The dataset was recorded with a qVGA resolution event-based sensor

and features a square and bars moving up and down in front of the stationary event camera. This dataset is denoted "Bar-Square" data. In the recording, the bars are always moving perpendicular to the true direction of motion, meaning that the ARMS algorithm should achieve an accurate estimate of the true flow in most cases.

This dataset was used to test more than 60 different hardware design configurations. The number of spatial windows (η), maximum size of the spatial windows (W_m), and the number of parallel accelerators (P) were all varied and the results evaluated. The direction estimation accuracy, throughput, throughput speedup, FPGA resource utilization, and estimated power usage were all collected to evaluate trade-offs in the design configuration selections.

4.1.1.1 Direction Estimation Accuracy

The Bar-Square data involves motion of the scene in only one direction—either moving upward or downward. This means that for any of these movements, an ideal optical flow algorithm should output a direction distribution with one peak and with a standard deviation of zero. This is used to quantify the performance of the different implementations of the flow algorithms. Thus, the direction estimation error is quantified as the standard deviation of flow angle results across all the events. Larger standard deviation of angles indicate larger error in correcting the direction of motion from normal to true direction. A low standard deviation indicates that the true flow calculated has only one primary direction of motion, which is what is expected from the dataset. It is important to note that the configuration that provides the best standard deviation for this visual scene, will not necessarily provide the best results in all scenes, but it does allow comparison of accuracy between multiple algorithms and configurations. The values of W_m and η which provide the optimal results will vary based on sensor resolution and visual scene activity.

Fig. 4.1 provides an example of hARMS flow correction when the scene is moving up (top) and down (bottom). The local-flow results are generally normal to the moving edge and are noisy in both magnitude and direction. The corrected hARMS flow results correctly capture the true direction of motion. This behavior is seen in the direction distribution histogram, which shows strong peaks at 90 deg and -90 deg as opposed to the local flow,

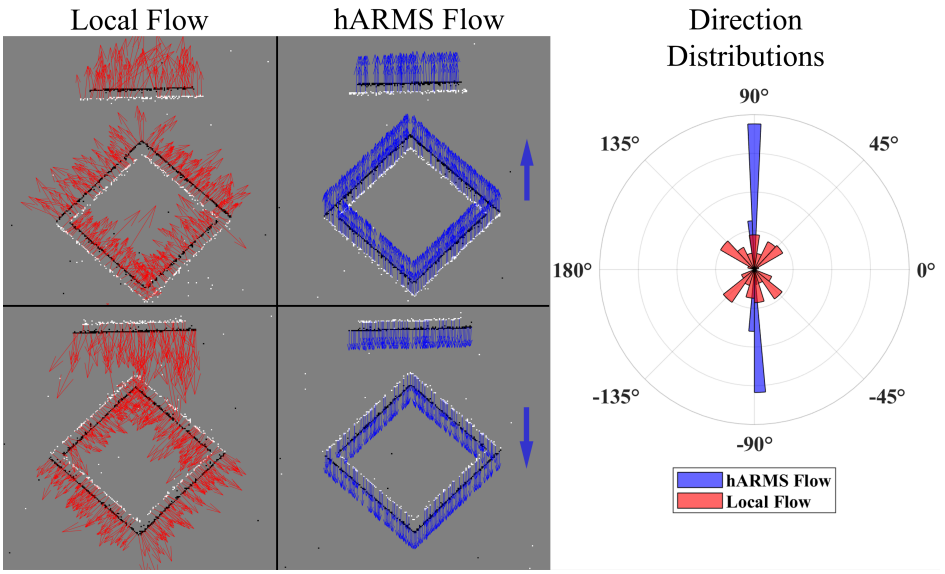

Figure 4.1: Bar-Square results are shown for the two directions of motion, up (top) and down (bottom). Local-flow results are shown using red vectors and hARMS output is show in blue.

which has multiple erroneous peaks in direction frequency. The hARMS results also show a more uniform magnitude of flow across the sensor frame.

The direction standard deviation results across multiple values of η for the ARMS, fARMS, and hARMS algorithms are also evaluated. These results are shown in Fig. 4.2 which uses values of 320 for W_m and 1000 for N. The value of N was chosen to ensure that all of the true-flow events within the temporal window set by τ are considered. A significant improvement in direction estimation accuracy for fARMS and hARMS is observed when compared to the original ARMS algorithm. This behavior is a result of the optimizations included in the fARMS algorithm. The use of multiple events at the same pixel within the temporal window, as made possible by the ring buffer realization of the RFB, likely improves performance due to the occurrence of this behavior along strong edges where local-flow estimates are most accurate. The fARMS and hARMS results are almost identical for all window sizes, with only slight variance due to the quantization of inputs to the hARMS

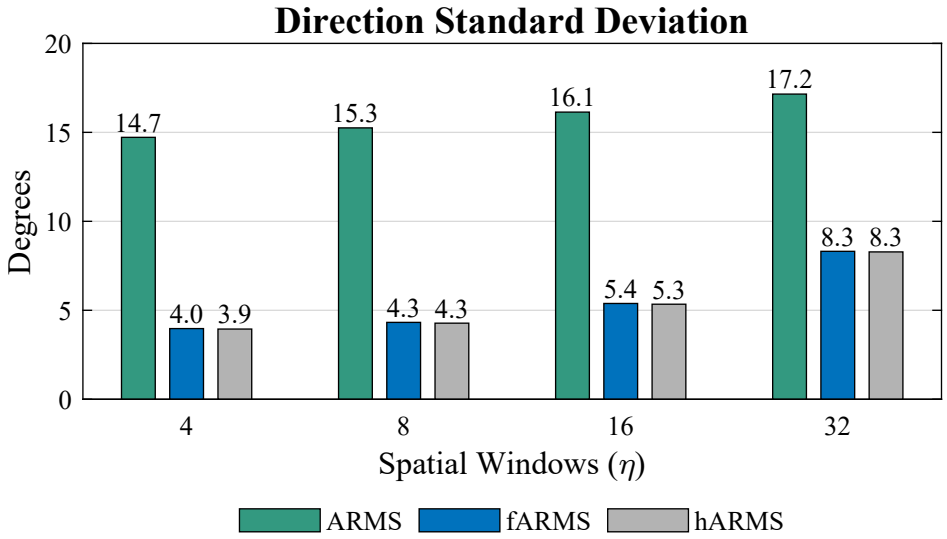

Figure 4.2: Flow direction estimates standard deviation for different design configurations. The value of W_m is constant at 320 for all results shown and N is fixed at 1000.

accelerator. The value of P has negligible impact on direction estimation accuracy for all tested values of η. This validates the use of a single input buffer of recent-flow events rather than individual buffers for each of the last P true-flow events.

The direction estimation accuracy was also evaluated as a function of RFB length, N. While reducing N improves the throughput of the design, it also reduces the number of recent local-flow estimates that can be used for estimating the true direction. Because of this reduction in data, the standard deviation of angle estimates increases as the buffer length is reduced. This behavior is shown in Fig. 4.3. There is also a certain point beyond which increasing N will not improve accuracy because all additional events will be filtered by the temporal window constraint. This is seen in Fig. 4.3 between buffer lengths of 1000 and 2000 where there is no change in accuracy. Despite the increase in standard deviation as N decreases, the accuracy was below the of the original software ARMS algorithm. This proves that even with a significant reduction in the amount of data considered, the hARMS and fARMS designs can achieve accurate results.

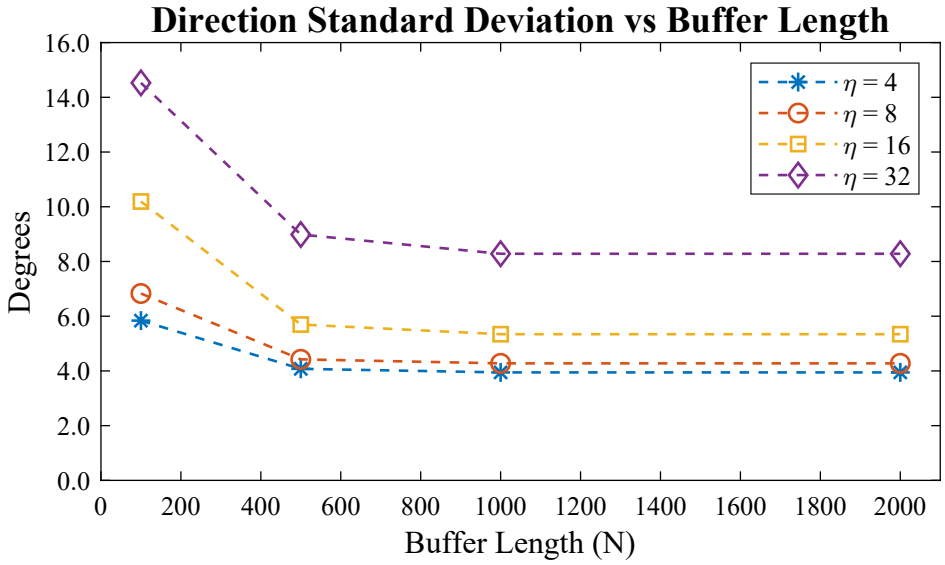

Figure 4.3: hARMS standard deviation results for different values of η as N changes. Each point represents the average standard deviation across all values of P tested at that configuration.

4.1.1.2 Throughput

Design throughput is measured in true-flow events per second (evt/s). As many events will not have viable local-flow results such as those generated due to noise, the event rate coming from the sensor could be significantly higher than the maximum design throughput without overwhelming the design. However, we consider only the throughput of the hARMS design after filtering and prepossessing of events occurs. This allows for the evaluation of the worst-case scenario where every event generated by the sensor is a valid true-flow event. In Section 4.1.3, a real-time evaluation of the hARMS architecture is provided, which includes information regarding total events and true-flow events in each dataset considered.

Fig. 4.4 shows the throughput results for fARMS and hARMS with varying P for different number of spatial windows (η). The results shown were generated for a selected benchmark configuration where $W_m = 320$ and $N = 1000$. The figure shows that the throughput of the fARMS software design decreases as the number of spatial windows is increased.

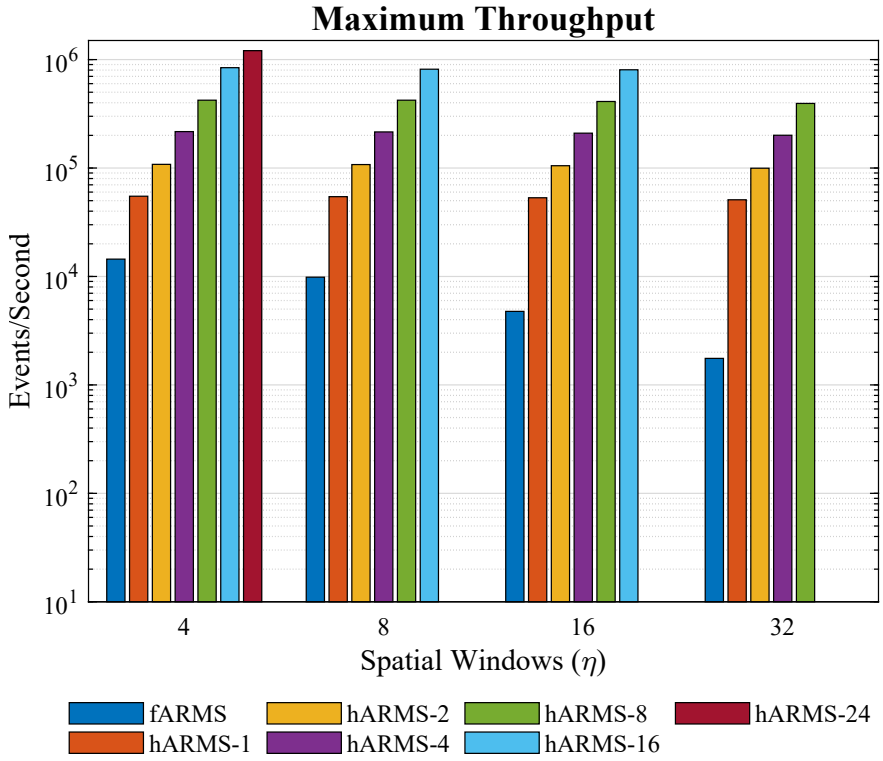

Figure 4.4: Maximum throughput results using qVGA Bar-Square dataset. Hardware configurations are denoted as hARMS-P.

However, as expected, the throughput for each hardware configuration is nearly constant across varying numbers of spatial windows, with only small decreases as the number of windows increases. This behavior is due to the streaming architecture and fully unrolled window searching implemented in hardware. The small decreases in throughput that are observed as η increases are a result of the number of dividers being limited to four per averager. The highest throughput achieved with $N = 1000$ is 1.21 Mevt/s when $\eta = 4$ and $P = 24$.

Higher throughput speedups were achieved for larger values of η due to the decreasing performance of the fARMS baseline with large numbers of spatial windows. That poor performance results from the need for sequential iteration through the windows in software. The parallel nature of hARMS and the streaming architecture eliminates this bottleneck

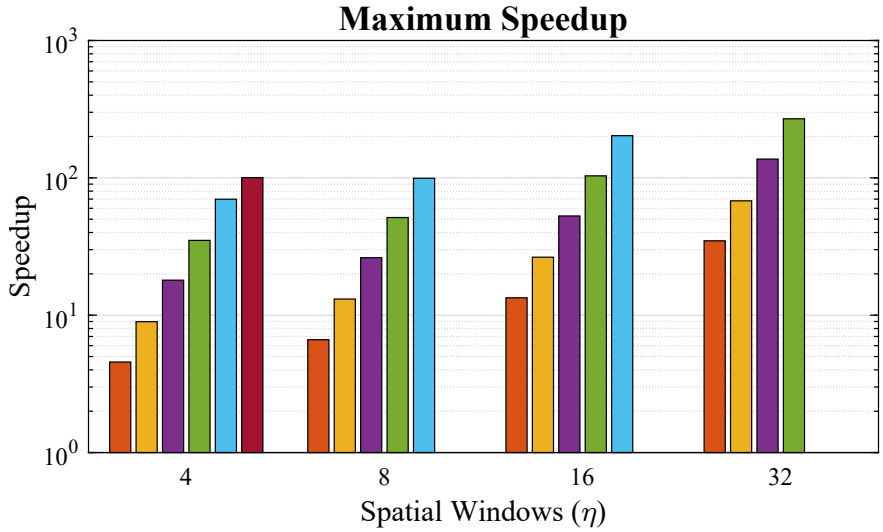

Figure 4.5: Maximum speedup results using Bar-Square dataset. The speedup is measured over the optimized fARMS design outlined in Section 3.1. W_m is equal to 320 in all cases and N is 1000 for all cases.

and allows for substantial speedup over the software baseline. As shown in Fig. 4.5, speedup ranges from 4.6× for $P = 1$ and $\eta = 4$ to 269.2× for $P = 8$ and $\eta = 32$.

The value of W_m only affects the hardware implementation of the predetermined edges in the $tagLUT$ module, and therefore has no impact on the latency or throughput of the design. The final parameter that has significant impact on the throughput of the design is N. Its value was set at 1000 for the primary benchmark because analysis of the Bar-Square dataset showed that a RFB with length 1000 is sufficient to capture all the recent-flow events within the τ temporal window of 5 ms used. However, in some cases the length of the RFB will need to be larger or smaller based on use case and desired throughput and accuracy. Fig. 4.6 shows the relationship between N and the throughput of one hARMS accelerator core. When N is greater than 1000, the expected trend is observed with throughput decreasing as N increases. Throughput decreases slightly as η increases for a given value of N. This is caused by the limitation of four hardware dividers per averager in the hARMS core, making repeated calls to division resources necessary for higher numbers of windows. At smaller values of N we

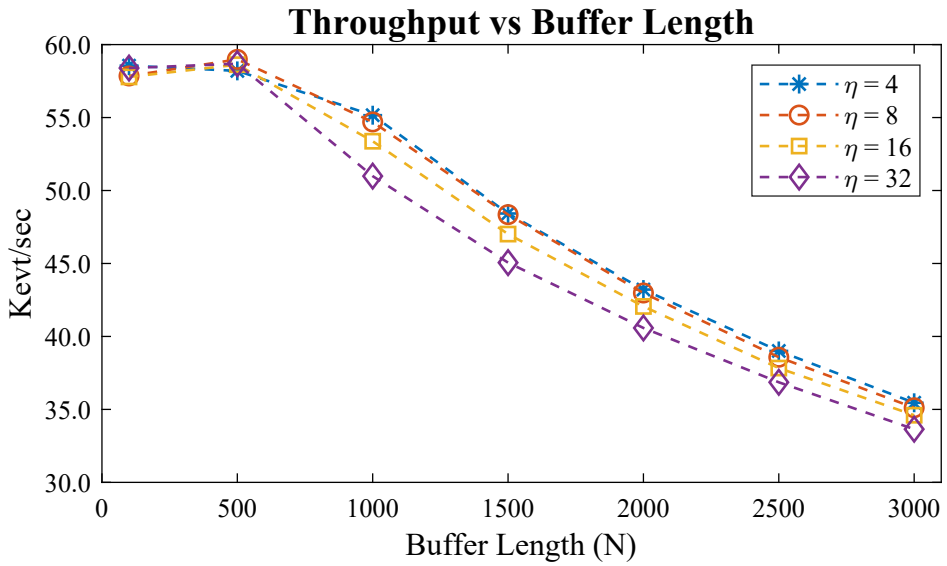

Figure 4.6: hARMS throughput measured using Bar-Square dataset for varying buffer lengths and numbers of windows. P is fixed at 1 for all cases.

observe unexpected behavior, with all configurations having nearly identical flat values for throughput. This is due to the latency of the data transfer between the PS and PL regions, which for small buffer lengths becomes the dominant latency during the hardware function execution. This latency could be removed by providing a direct connection between the event-based vision sensor and the PL region of the SoC. This would result in a significant increase in throughput across all configurations, but it would limit the versatility and configurability of the design as well as require an FPGA implementation of local flow. For these reasons it was not included in the hARMS architecture.

4.1.1.3 Resource Utilization

FPGA resource utilization is evaluated based on the implemented design for each configuration. LUT and FF usage are of primary interest in evaluating how the design scales as parameters change. BRAM utilization is evaluated in relation to the value of N, while DSP utilization remains at zero across all configurations. BRAM usage is independent of

all parameters except N as it is only used for the data motion network and the RFB. The volume of data transferred by the DMA controller changes with P, this however does not impact the BRAM usage required for the interface meaning that BRAM scales only with N.

The LUT and FF usage as a function of P is shown in Fig. 4.7 and Fig. 4.8 respectively. Both resources show linear scaling as the value of P increases. The rate of scaling is dependent upon the number of spatial windows used. Table 4.1 shows the per parallel accelerator core resource usage for each of the values of η tested. This linear scaling is expected due to the limited opportunity for resource sharing between accelerator cores. Only the DMA, event RAM, and result RAM resources can be shared between accelerator cores.

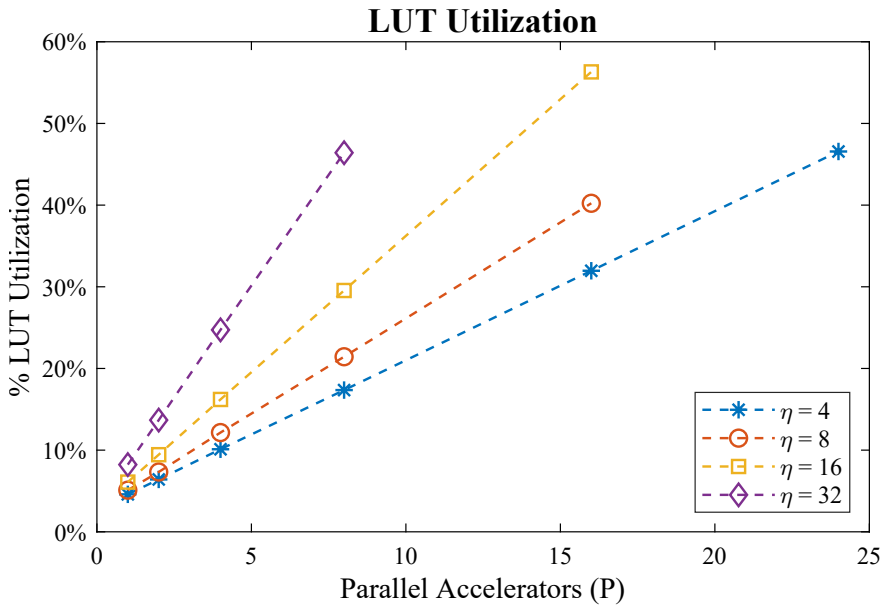

Figure 4.7: LUT utilization vs P for different number of spatial windows. Utilization is represented as a percentage of the 218600 total LUT available on the Zynq-7045 SoC used.

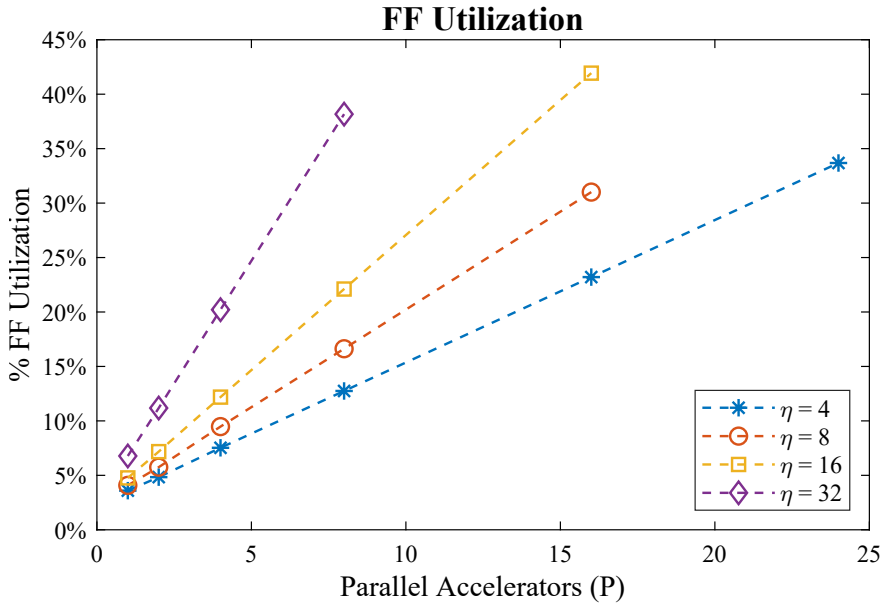

Figure 4.8: FF utilization vs P for different number of spatial windows. Utilization is represented as a percentage of the 437200 total FF available on the Zynq-7045 SoC used.

Table 4.1: Resource usage per accelerator core for various η implemented on Zynq-7045 SoC.

Spatial Windows (η)	**LUT/core**	**FF/core**
4	3959 (1.81%)	5704 (1.30%)
8	5073 (2.32%)	7815 (1.79%)
16	7291 (3.34%)	10848 (2.48%)
32	11853 (5.42%)	19627 (4.49%)

BRAM utilization as a function of N is shown in Fig. 4.9. It shows a baseline utilization of 3.5%, which is required for the DMA network and a relatively small RFB. As the buffer length increases the BRAM utilization increases. The increase takes on a step-like characteristic because although the absolute size of the RFB increases linearly with N, a BRAM block can be considered utilized without its whole memory capacity being used. Overall, BRAM usage is very small compared to the LUT and FF utilization. The hARMS architecture is not dependent on storing an event frame with the same size as the sensor resolution like most existing hardware and software designs. This results in more efficient use of memory in the PL fabric as only the most relevant events are stored.

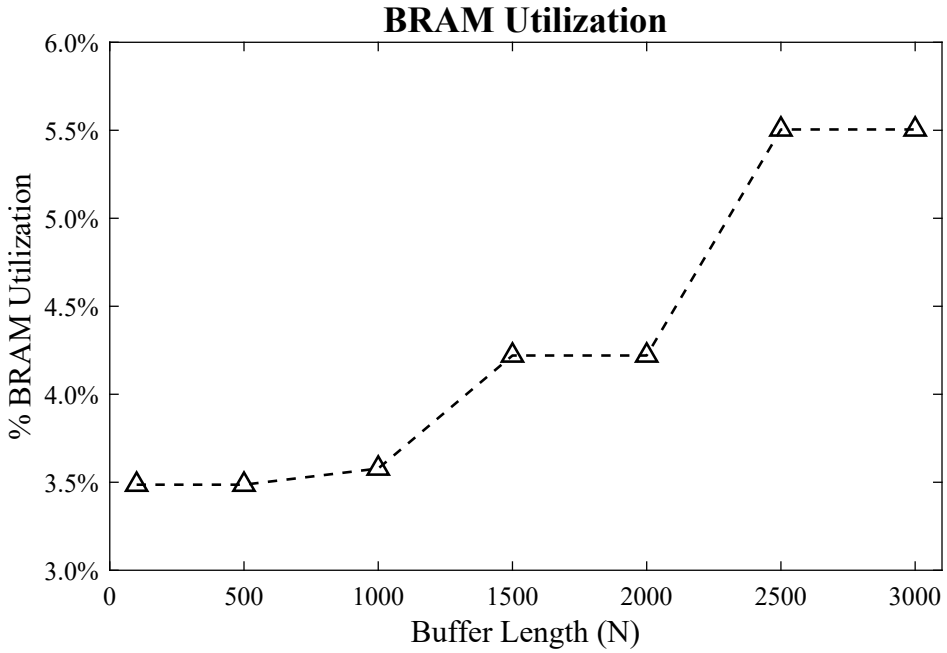

Figure 4.9: BRAM utilization vs N for different number of spatial windows. Utilization is represented as a percentage of the 545 total 36Kb BRAM blocks available on the Zynq-7045 SoC used.

4.1.1.4 Estimated Power

Power limitations often accompany embedded systems. One advantage of neuromorphic hardware is that it often operates at low-power consumption. To maximize the benefit of sensor power efficiency, it is important to consider the processing power requirements as well. Fig. 4.10 shows how the estimated dynamic power requirements of the FPGA change with the value of P. These power estimates are generated from Xilinx's Vivado design tool. Like FF and LUT resource utilization, estimated power consumption scales linearly with P.

Overall dynamic power consumption ranges from 0.305 to 1.310 Watts for configurations implemented. Dynamic power for low values of P is dominated by clocking and clock generation which accounts for 200 mW or more of dynamic power depending on the number of windows used. In some cases, this power consumption could become prohibitive for power constrained embedded platforms. To address this concern, steps could be taken to improve

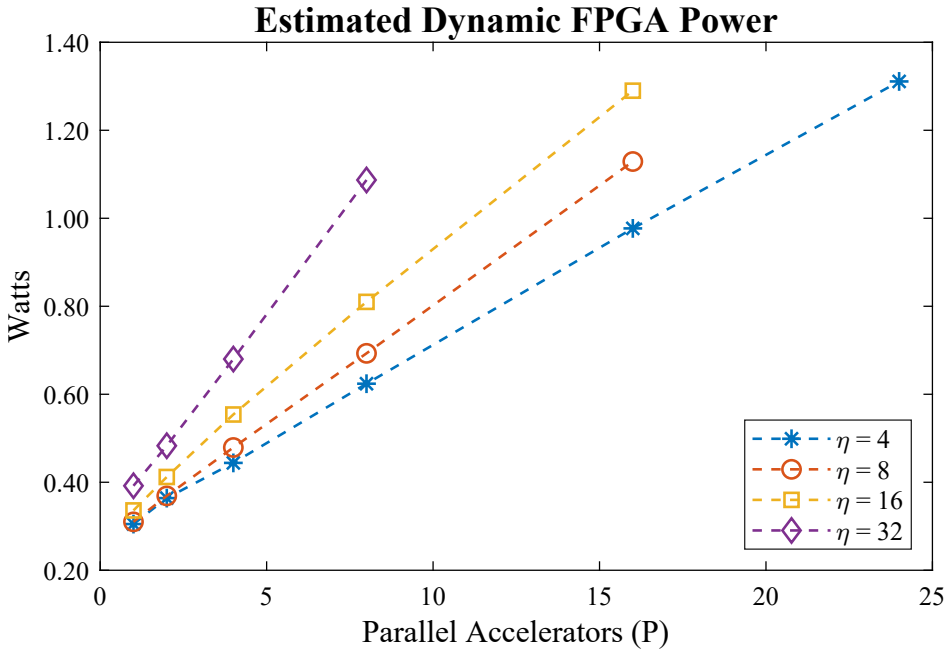

Figure 4.10: Estimated dynamic FPGA power consumption vs P for different values of η. Estimates are obtained from Xilinx Vivado after bitstream generation is completed. Results are for implementation on the Zynq-7045 SoC and do not include the power consumption of the ARM processing system. N is 1000 for all estimates.

power efficiency such as reducing the size of P and η. Reducing the hARMS operating frequency could also yield reductions in power, however at the cost of reducing the throughput of the design. This research focuses on optimization of the design for latency and throughput.

4.1.2 hARMS on Real-World Datasets

Based on the results from the Bar-Square dataset, we implemented the hARMS configurations to compute aperture robust optical flow on more complex real-world scenarios such as the DAVIS dataset [23], the MVSEC data [29] and a VGA resolution recording [1]. The results for these are presented in the next sections. Because fARMS and hARMS produce flow results with only small differences due to quantization, only hARMS results will be included in the following sections.

4.1.2.1 Dynamic Rotation Dataset

To ensure that the redesigned algorithm used for fARMS and hARMS maintains accuracy in dynamic scenes, the DAVIS dataset presented in [23] was used. This dataset provides the event stream from the 180×240 px resolution DAVIS along with timestamped grayscale images. The dataset also includes inertial measurement unit (IMU) data collected at a rate of 1000 Hz, which provides the angular velocity of the camera as the scene is being recorded.

The dataset recording selected is the dynamic rotation scene as it allows for easier mapping from optical flow to the angular velocities from the IMU. In this scene the DAVIS is rotating along its axes while recording an office scene. The dynamic nature of the scene can be seen in Fig. 4.11, which shows the local-flow direction estimates (left) and the hARMS flow direction estimates (right) for three distinct directions of motion. It is observed that even in a dynamic scene, the hARMS design performs well, producing the expected true direction of motion as its output, even from noisy local-flow direction estimates. For comparison to the software design study performed in [1], the same algorithm parameters were used, therefore, the hARMS configuration used to process the DAVIS dataset was ($W_m = 100$, $\eta = 10$, $\tau = 5ms$, $P = 16$, $N = 1500$). The size of N was intentionally set below the maximum required size to avoid losing events in the temporal window. This was done to show that even with a buffer length of less than half the minimum size to capture all relevant events, the fARMS algorithm and hARMS accelerator can still provide accurate output flow in dynamic scenes.

The IMU data provided for the dynamic rotation scene was used to quantify the accuracy of the hARMS results in this dynamic scene. The flow velocities in the x and y directions were compared to the angular velocity measurements from the IMU. Both results are shown in Fig. 4.12, which shows high correlation between the hARMS results and the ground truth. The hARMS results achieve a correlation value greater than $R = 0.93$ for both the x and y flow estimates.

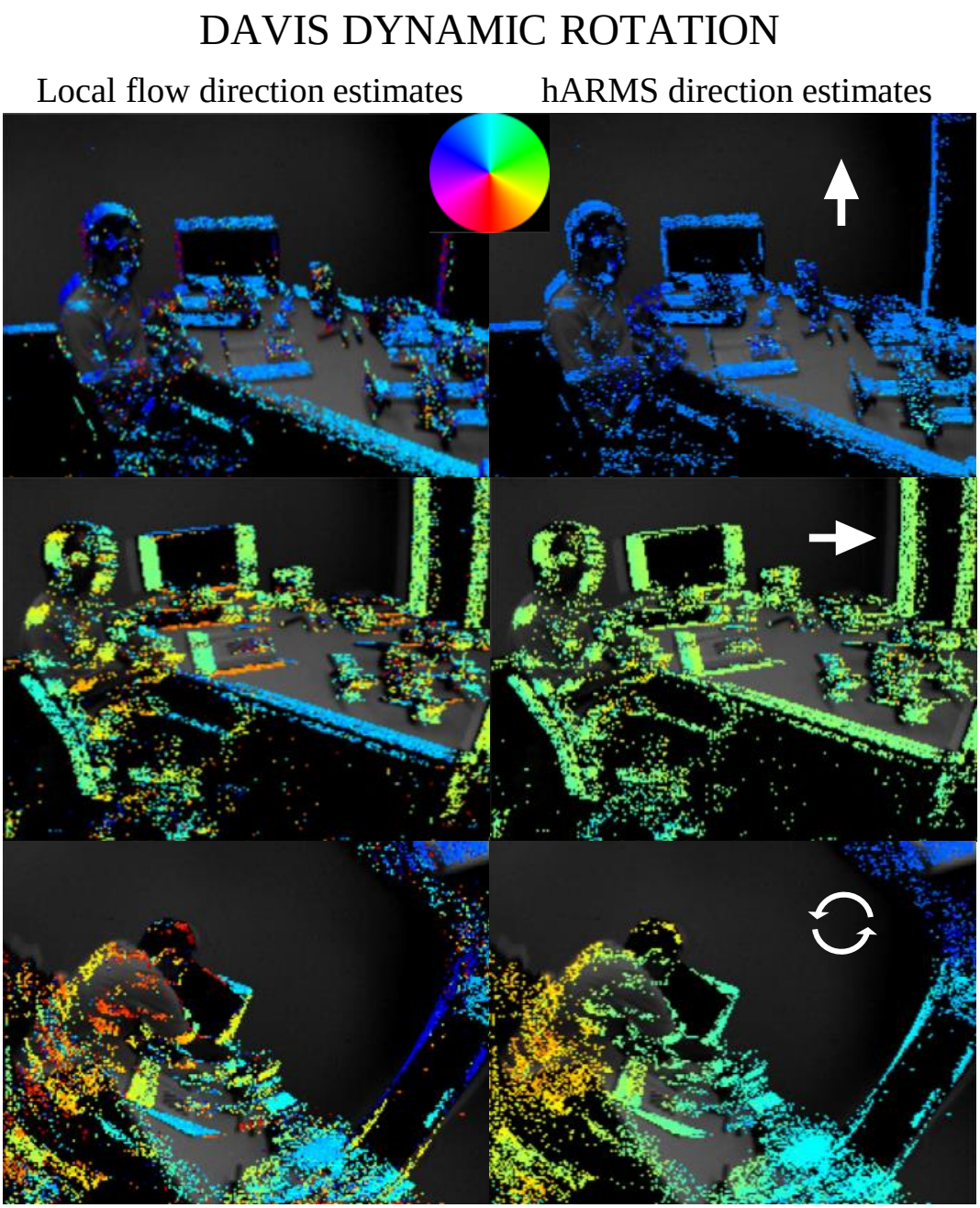

Figure 4.11: Qualitative results showing correction of local-flow estimates using the hARMS design. The panels show local and hARMS flow direction estimates for events recorded using DAVIS. The events are overlaid on grayscale images captured using the DAVIS.

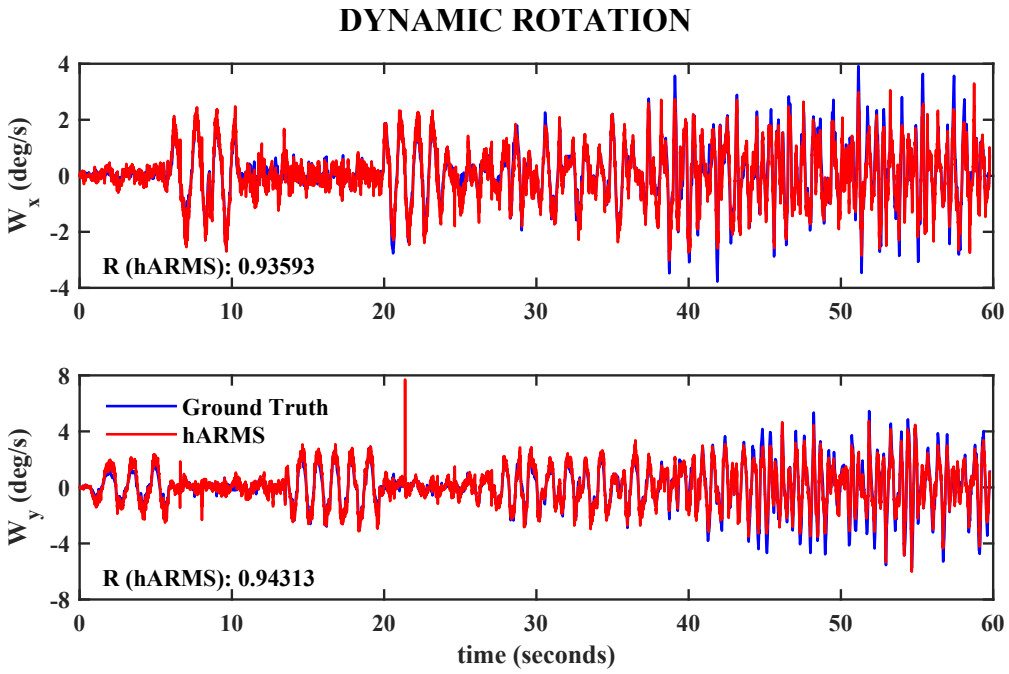

Figure 4.12: Comparison of hARMS results with IMU ground truth for dynamic rotation DAVIS dataset.

4.1.2.2 MVSEC Dataset

For further verification of the design for optical flow estimation in real-world environments, hARMS is applied to scenes in the Multi Vehicle Stereo Event Camera (MVSEC) dataset [29]. The MVSEC dataset utilizes an array of sensors to provide dense ground truth optical flow at specified frame intervals. The same approach used in [1] is followed to map the asynchronous optical flow outputs from the hARMS implementation to the frame based ground truth. The hARMS configuration used to process the dataset was (W_m = 100, η = 10, τ = 5ms, P = 16, N = 1500). The hARMS results are plotted with the ground truth for four of the MVSEC dataset recordings as shown in Fig. 4.13. The hARMS results successfully correct the local flow and closely follow the ground truth provided. This is especially true for the indoor flying scenes where the hARMS results closely follow the ground truth throughout the recording. The hARMS flow does struggle more with the outdoor day

recording where the hARMS output is much noisier than the ground truth. This is especially noticeable during 22 to 32 second interval when the hARMS results overshoot the Vy ground truth and undershoot the Vx ground truth with high noise in both cases. As was observed in the original software realization presented by [1], this loss of accuracy can occur in areas with poor local-flow estimates or rapid changes in direction for which the ARMS flow cannot quickly correct.

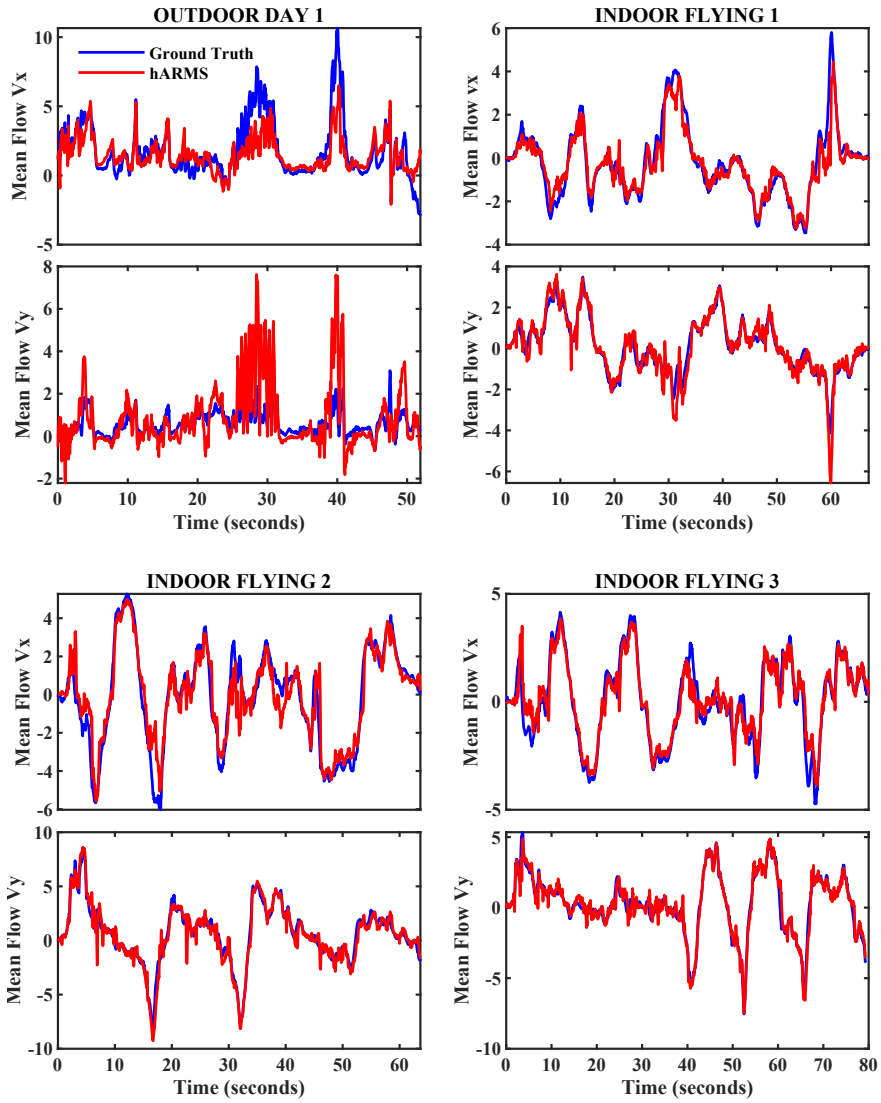

Figure 4.13: Comparison of hARMS flow results with the provided ground truth for four MVSEC scenes.

4.1.2.3 Pendulum VGA Dataset

To verify the performance of the hARMS design for multiple directions of motion and occluded objects within a scene, the pendulum dataset introduced in [1] is used. This scene uses a VGA (640×480 px) resolution sensor to record two pendulums of the same length oscillating in front of the sensor. One pendulum is placed further from the sensor such that it will appear smaller as it passes behind the other pendulum. For this dataset, the following hARMS configuration is used for processing: ($W_m = 50$, $\eta = 5$, $\tau = 5ms$, $P = 16$, $N = 2000$). A smaller value of W_m was chosen based on the observation from [1] that such selection improves flow results in the presence of occlusion.

Fig. 4.14 shows the results of hARMS processing on the pendulum dataset. At 0 ms in the sequence, the pendulums are swinging towards each other and hARMS produces accurate flow estimates for both objects. In the next 40 ms the further pendulum nears the closer and the flow estimates on the leading edge begin to erroneously assume the direction of motion of the closer pendulum. This continues until the further pendulum is completely occluded by the closer pendulum at 80 ms. The same behavior is then observed as the two pendulums separate. The erroneous flow direction estimates are quickly corrected as the distance between the two pendulums increases. This behavior matches that of the original software algorithm and shows that momentary errors in direction due to occlusion are quickly rectified as objects separate in the scene.

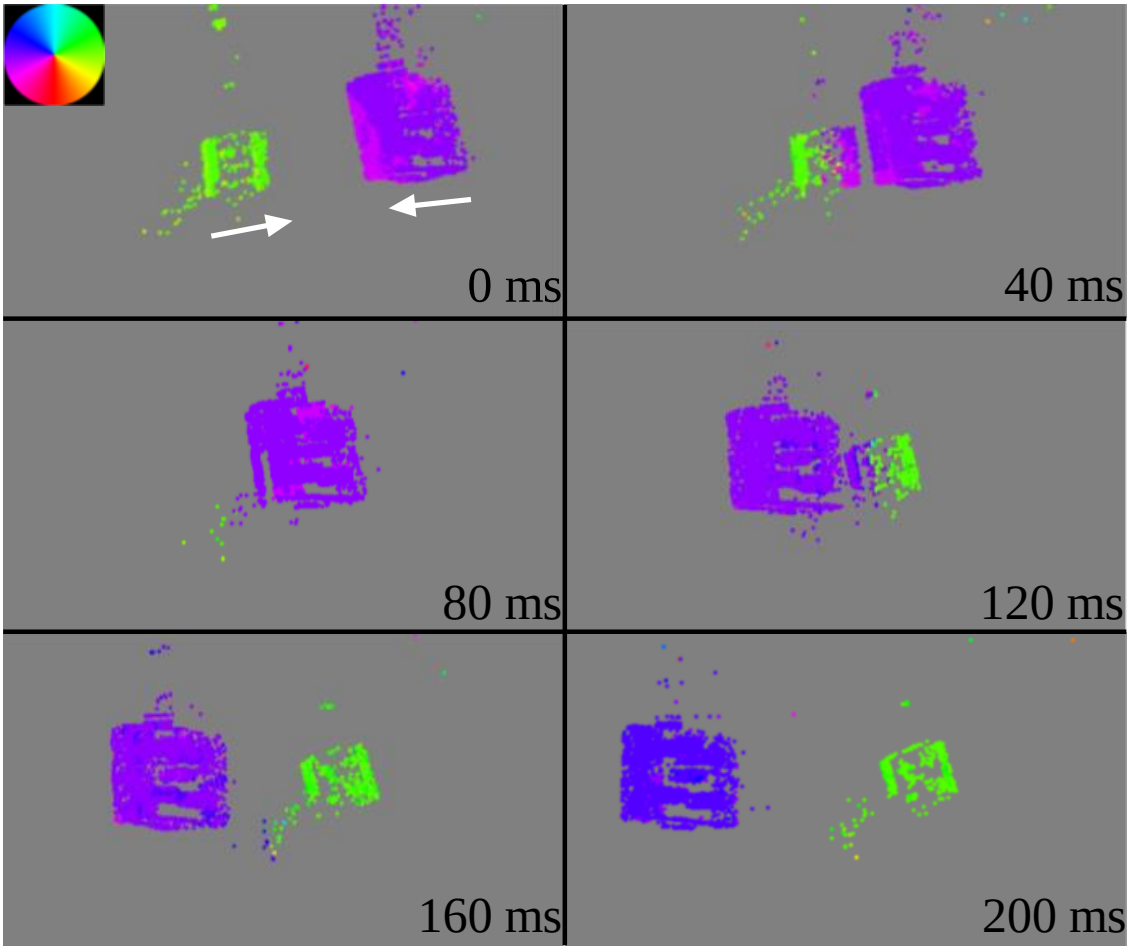

Figure 4.14: hARMS results for crossing pendulums at different visual depths. The event frames shown are accumulated over 20 ms of motion and only the relevant portion of the VGA sensor frame is displayed.

4.1.3 Performance Comparisons

The following sections provide real-time performance analysis and comparisons for the fARMS and hARMS designs. Performance evaluations are made for fARMS on both embedded and desktop grade platforms, while hARMS is restricted to its intended use case on embedded platforms. Performance is evaluated across a range of datasets and sensor resolutions.

4.1.3.1 Embedded Performance

Evaluating the performance of an asynchronous, event-based algorithm for real-time operation can be difficult due to the inherent dependency on the event rate at any moment in time. Scenes with high activity, and therefore high event rates will require higher algorithm processing throughput to achieve real-time operation. For this research, real-time operation is considered to be the case where the fARMS or hARMS compute rate exceeds the true-flow rate of a specific dataset, where true-flow rate is the number of events per second that require a true-flow calculation to be performed.

The dependence of hARMS and fARMS throughput on the buffer length, N, adds further difficulty to making performance comparisons across different datasets. Higher throughput can be achieved by reducing the input buffer length, however, this artificially reduces the amount of data available to correct the local-flow direction. To avoid this the minimum buffer length required to capture all of the relevant events within the temporal window τ is chosen. This requires pre-evaluation of the dataset to determine the buffer length, which is equal to the maximum number of true-flow events within a $\tau = 5\ ms$ window. In a real-time application where the events are not pre-recorded, the effective buffer length can be dynamically changed in based on the event rate as long as it does not exceed the initial value of N used for hARMS configuration. However, for the purpose of evaluation, the advance knowledge of the dataset is used to select the worst case buffer length for each dataset. That buffer length is then used for processing of the entire set.

For evaluation we chose the best performing configuration on the Bar-Square trivial motion dataset, which used $\eta = 4$. While the use of four spatial windows may not be optimal in all cases, it proved best for a singular direction of motion and is therefore used in all benchmarks to maintain consistency across datasets. The value of W_m does not impact the throughput of the design and can be adjusted based on the sensor resolution and operating environment. For the embedded benchmark testing it was kept at 160. The value of P was set to 16 for all hARMS tests due to its high throughput and reduced resource footprint compared to the 24 core configuration that has high throughput, but can only be utilized when η is four. The embedded tests use the same plane-fitting local-flow method as in [1]

and in the prior experiments for consistency. However, in applications where the local-flow computation could become a bottleneck, we propose the use of a more computationally efficient method for regularizing the timing of events called Savitzky-Golay plane-fitting presented in [25].

The results of embedded performance benchmarking are shown in Table 4.2. Real-time performance is achieved using fARMS for two of the datasets, however, as expected most fall short of real-time operation. The hARMS architecture is, however, able to easily achieve real-time performance across all datasets tested. The dataset for which real-time performance is most difficult is the dynamic rotation set. This set has by far the highest average true-flow event rate of all datasets evaluated and requires the largest RFB. Despite this, the hARMS compute rate exceeded the true-flow rate by 2.35× using the defined benchmark parameters. It is also important to note that flow accuracy can be maintained even with a more than 50% reduction in the buffer length, as shown in Section 4.1.2.1. Leveraging that finding could allow for even more comfortable achievement of real-time processing. The results on the other datasets show that not only can real-time performance be achieved, it can also be achieved with reduced resource utilization depending on the scene. Observing the true-flow rates in Table 4.2 and the throughput results discussed in Section 4.1.1.2, it can be seen that real-time operation can be achieved with as little as one or two hARMS cores for most datasets, depending on the configuration.

Another takeaway from the results shown in Table 4.2 is the independence of fARMS and hARMS throughput from sensor resolution. In fact, the lowest resolution sensor has the

Table 4.2: fARMS and hARMS throughput performance comparison for various dataset scenes on Zynq-7045 embedded platform. Real-time operation indicated in bold.

Dataset	Sensor Resolution	Total Events	True-flow Events	Recording time (sec)	True-flow Rate (Kevt/s)	Buffer Length	fARMS Rate Kevt/s)	hARMS Rate (Kevt/s)
Dynamic Rotation	240×180	71.32e6	12.97e6	59.77	217.00	3286	5.1	**509.1**
Bar-Square	304×240	1.25e6	530933	5.80	91.54	906	15.3	**840.0**
Outdoor Day 1	346×260	20.0e6	3.71e6	52.86	70.19	1267	15.3	**753.2**
Outdoor Night 1	346×260	20.0e6	3.80e6	53.13	71.52	2209	9.6	**612.2**
Indoor Flying 1	346×260	12.0e6	1.65e6	69.02	23.91	519	**43.6**	**825.9**
Indoor Flying 2	346×260	20.0e6	2.30e6	70.81	32.48	904	26.2	**803.9**
Indoor Flying 3	346×260	20.0e6	2.18e6	83.91	25.98	710	**35.4**	**811.7**
Pendulum	640×480	4.82e6	618312	5.15	120.06	1853	8.9	**648.1**

slowest compute rate, while higher resolution datasets could be processed much faster. This indicates that the most important factor in determining the fARMS and hARMS throughput performance is the true-flow event rate, which directly impacts the required buffer length. That true-flow event rate is directly and most significantly impacted by the visual scene dynamics. The sensor resolution is a secondary contributor to an increased true-flow event rate and therefore to changes in fARMS and hARMS throughput.

4.1.3.2 Desktop Performance

The optimizations used in the fARMS algorithm provide a significant decrease in the computational complexity of the algorithm and allow for real-time performance on an embedded platform in some limited cases. Although event-by-event flow results will vary between the original ARMS and fARMS algorithms due to these optimizations, the previous sections have shown that the overall flow results generated by fARMS have equivalent or better accuracy across a wide range of visual scenes when compared to ARMS. Having demonstrated comparable flow accuracy, we now compare the throughput of both software algorithms on a desktop grade platform. As in [1], an Intel E5-1603 processor is used to collect the performance results, which are shown in Table 4.3. Both fARMS and ARMS use $W_m = 320$ and $\eta = 4$, while the fARMS configuration uses the specified buffer length as determined for the embedded platform tests. Real-time operation is defined in the same way as for the embedded tests.

The results in Table 4.3 show that real-time performance is achievable with the fARMS algorithm for all but two datasets. The ARMS algorithm, however, is far from real-time in all cases with the specified parameters. This demonstrates that the fARMS algorithm yields significant throughput performance improvements over the original realization presented in [1]. Although these software results show real-time performance in many cases, they still fall significantly short of the throughput obtainable by the hARMS architecture. This shows that the hARMS architecture can yield performance improvements even in non-embedded computing environments.

Table 4.3: Original ARMS vs faster ARMS (fARMS) throughput performance comparison for various dataset scenes. Real-time operation shown in bold.

Dataset	**Buffer Length**	**ARMS Compute Rate (Kevt/s)**	**fARMS Compute Rate (Kevt/s)**
Dynamic Rotation	3286	1.77	60.85
Bar-Square	906	2.33	**133.439**
Outdoor Day 1	1267	1.693	**96.527**
Outdoor Night 1	2209	2.137	**210.036**
Indoor Flying 1	519	1.646	**207.27**
Indoor Flying 2	904	1.775	**94.343**
Indoor Flying 3	710	1.833	**165.234**
Pendulum	1853	1.683	81.205

4.2 Event-Stream Compression Evaluation

The following sections provide evaluation the FBC method introduced. The impact of changing the timing parameters on the evaluation metrics established in Section 3.3.4 is considered first. An example of how the reconstruction performs over time when applied to a dataset is then shown. Finally, FBC is applied to a variety of real-world datasets and the results are presented and discussed.

4.2.1 Parameter Characterization

To characterize how changes in the timing parameters impact the evaluation metrics, the Bar-Square dataset introduced in Section 4.1.1 is used. The fARMS algorithm with Savitzky-Golay plane-fitting local flow is used to compute the optical flow in order to achieve denser flow results. Of the 1.26 million events in the dataset, a valid flow estimate is found for 1.06 million of them. This means that at least 200 thousand events must be sent from this event stream, corresponding to a maximum possible ER of 0.84.

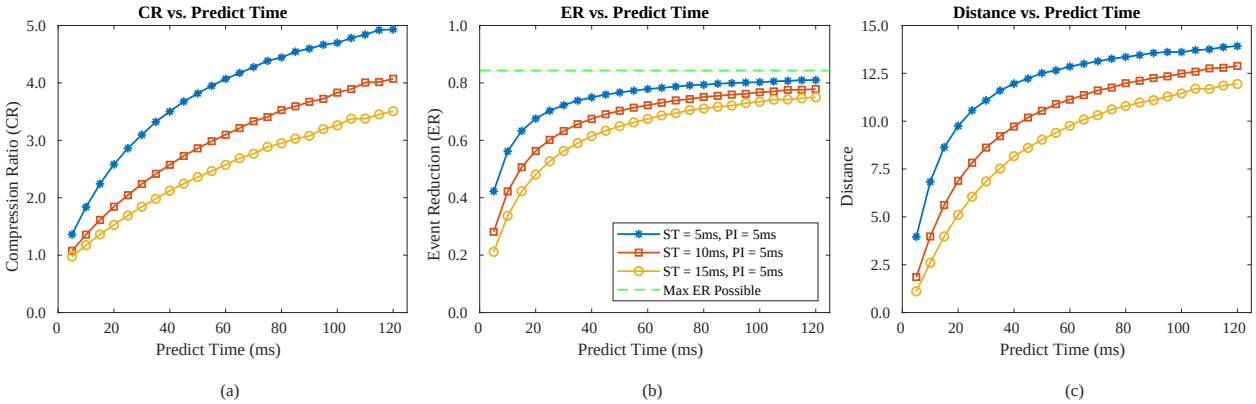

Figure 4.15: Characterization of predict time's impact on evaluation metrics.

Fig. 4.15 shows the changes in CR, ER, and distance as the value of PT changes for three selected values of ST. From these plots it is observed that as CR and ER increase, the distance between the reconstructed event stream and the original stream increases monotonically. This is the predictable result of less data being transmitted and longer duration predictions being performed. The results also verify the expectation that a shorter send time results in a higher CR and ER. However, this increased compression comes at the cost of increased distance. The operating point can be selected based on the application and error tolerance in the reconstructed stream.

The selection of an appropriate value for PI is also an important design decision that must be made based on the application requirements for compression and reconstruction fidelity. It is important to note that the computational complexity of event prediction will increase as PI decreases due to increasing numbers of predictions per second. Fig. 4.16 shows how the distance and the number of predicted events changes as the value of PI varies. As PI is increased, the number of predicted events, which include events sent with no flow, decreases due to less frequent prediction of new events. Distance increases monotonically as PI increases, indicating that more events lowers distance overall by matching the original events well even if there are extraneous events in the reconstructed stream.

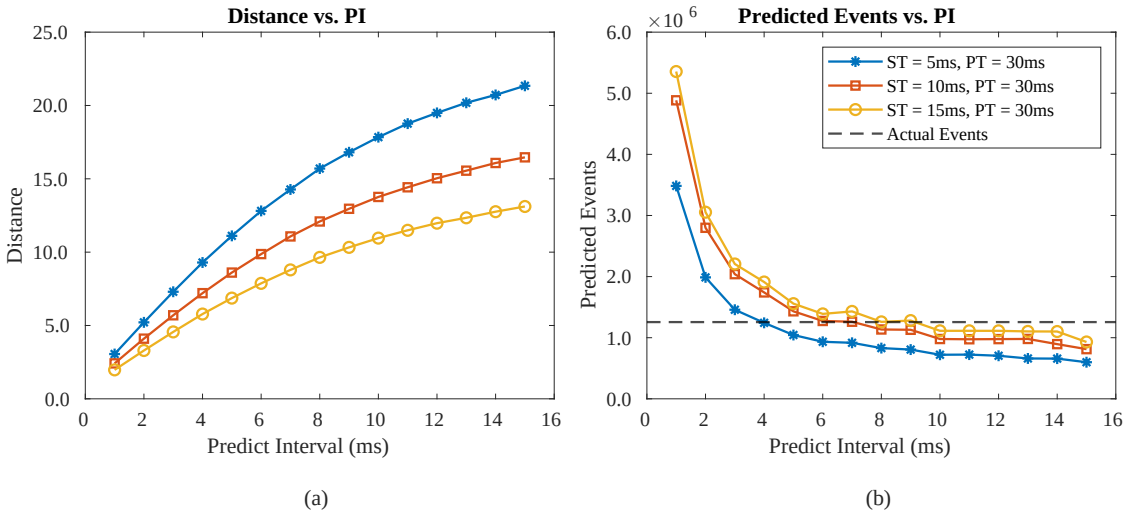

Figure 4.16: Characterization of predict interval's impact on distance and total number of events in the reconstructed stream (predicted events).

The effect of the vanishing events phenomenon discussed earlier can be observed in Fig. 4.16(b). The dashed line represents the actual number of events in the dataset. In the ideal case where no event predictions overlap, the number of predicted events should intercept this line when $PI = ST$. However, the actual intercept is at a lower PI due to the vanishing events. For example, when $ST = 5\ ms$ the actual intercept occurs at $PI = 4\ ms$. Depending on the end application, maintaining the same number of events in the reconstructed stream as the original stream may be more or less important. Based on the results in Fig. 4.16(b), the use of $PI = 0.8ST$ is a good choice for general applications of FBC.

4.2.2 Temporal Behavior

In addition to characterizing the impact of the parameters on the evaluation metrics, it's also important to evaluate how the FBC behaves over time while compressing an event stream. Fig. 4.17 shows the reconstruction performance over time compared to random removal of events. A 100 ms moving average is displayed to allow trends to be easily observed. Based on the performed parameter characterization, the configuration ($ST = 5\ ms$ $PT = 20\ ms$ $PI = 4\ ms$) is selected to achieve a high CR and ER while maintaining

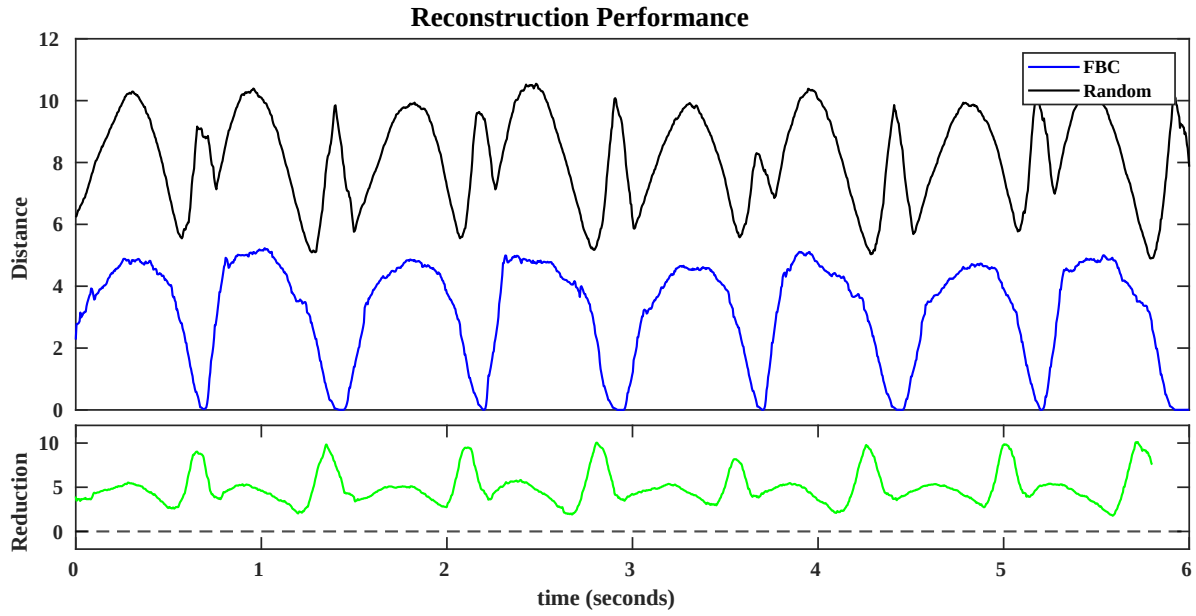

Figure 4.17: Reconstruction performance over time compared ER achieved with random removal of events.

a relatively low distance. The randomly reduced event stream is generated by randomly removing events from the stream in order to achieve the same ER as the FBC method.

Fig. 4.17 shows that the FBC distance is always less than the randomly reduced stream. Additionally, periodic behavior is observed due to the oscillation of the objects in the Bar-Square dataset. Local minima in FBC distance occur when the objects are changing direction. At these moments it is difficult for flow to be estimated, so most of the events are being sent and are therefore exact matches to the original stream. These periods where a low proportion of events get flow estimates could result in available bandwidth being saturated by the event rate. However, these moments are typically characterized by low overall event rates and are transitory by nature, meaning the overall system impact is temporary. The observation of these moments where most events are being sent validates the assumption that events without flow should be sent to enable proper event-stream reconstruction. On average, the FBC method has a 59.86% lower distance over the duration of the stream while also preserving more event data in the reconstructed stream.

4.2.3 Real-World Dataset Performance

Additional experiments were performed on various real-world datasets to evaluate the performance across different scene dynamics. The results of these experiments are shown in Table 4.4. These results were collected using the same FBC configuration used in Section 4.2.2. The results show that across varied datasets the FBC method achieves an average CR of 2.47, which corresponds to the removal of 62% of events on average. This results in an average distance of 4.97. These results were all found using the same optical flow parameters. The indoor flying 1 dataset has an unusually low compression ratio that is caused by the fact that only 5.3 million of the 12 million total events in that dataset received an optical flow estimate. The low percentage of events with flow is caused by a combination of factors such as the scene dynamics, flow computation parameters, and sensor noise. The shapes dataset has by far the highest distance of all recorded results. This is due to high event rate portions of the scene that also have dynamic motion such as rapid changes in direction, which produce poor flow estimates. The slider far dataset has a CR of 7.6% less than the shapes dataset, but has a distance that is 56.2% less. This is because the slider dataset has constant motion with no rapid changes in direction.

These results demonstrate that the FBC method is applicable to a wide range of visual scenes and types of motion. Even without scene-specific parameter tuning, the results show a low average distance with a significant reduction in data transmission requirements. This method can be coupled with an existing lossless compression technique for further com-

Table 4.4: FBC performance on real-world datasets.

Dataset	CR	ER	Distance
Shapes [23]	3.14	0.75	11.54
Bar-Square [1]	2.58	0.68	3.27
Outdoor Day 1 [29]	2.25	0.61	3.60
Indoor Flying 1 [29]	1.48	0.36	1.40
Slider Far [23]	2.90	0.72	5.05
Averages	**2.47**	**0.62**	**4.97**

pression. It is found in [16] that the slider dataset can be compressed with a CR of 3.19 using LMZA dictionary encoding. Because the output of the FBC method is simply another stream, it allows for the subsequent use of another method, such as LMZA, to enable further compression. This would enable further compression especially during the *sending* state when the raw event stream is passed through the FBC method and significant redundancy still exists. The extent to which further compression may be achieved would be dependant on the performance of the secondary method used to compress the output of the FBC. However, given that FBC uses a different paradigm than all other existing methods it is likely that the addition of an existing method could yield a significant increase in the CR achieved.

5.0 Conclusion

The asynchronous nature and high-temporal resolution of event-based vision sensors make them ideal for computation of optical flow in the visual scene. This optical flow can then be used for other vision related tasks such as object tracking, collision avoidance, and flow-based compression. However, for these tasks, the optical flow must be computed in real-time and often on small embedded computing platforms with limited processing power. To capitalize on the benefits of accurate optical flow from event-based vision sensors, the development of fast embedded-processing solutions is essential. Although both software and hardware solutions exist to calculate event-based optical flow, no previous solutions have achieved real-time, aperture-robust true-flow calculation on an embedded platform while also utilizing and maintaining the high temporal resolution of the sensor.

This research has introduced an optimized event-based optical flow algorithm called fARMS along with a novel hybrid acceleration architecture that fulfills those requirements. The fARMS and hARMS architectures were developed based on the ARMS algorithm presented in [1]. The algorithm was modified to be amiable to a more asynchronous, neuromorphic implementation in hardware by discarding the use of a continuous event frame and instead operating asynchronously on only a small history of relevant events. This modification not only improved flow accuracy, with a decrease in angle standard deviation of up to 73%, it also allowed for the achievement of real-time processing rates that are independent of sensor resolution. hARMS processing throughput of up to 1.21 Mevt/s was achieved, making it the fastest realization of event-based true-flow demonstrated in literature. Real-time speed was achieved for every dataset considered, with most datasets able to be processed in real time with low resource configurations of the hARMS architecture. Thorough analysis of the design space was performed to determine the relationship between the architecture parameters and the throughput, accuracy, resource utilization, and power of the resulting accelerator. Resource utilization scales linearly with the number of accelerator cores used, with FF and LUT utilization being the main constraints on design scaling.

Unlike previous works that achieve aperture robust flow ([20, 19, 28]), the hARMS architecture can operate in real time, fully utilizes the temporal resolution of the sensor, and operates only on temporal contrast events. When compared to the estimated resource utilization in [21], the hARMS architecture offers configurations capable of achieving real-time performance with fewer resources. The achieved throughput of 1.21 Mevt/s enables hARMS to nearly match the throughput achieved by the local-flow FPGA implementation in [2]. Matching this performance means that the design in [2] could be used to generate the local-flow input to the hARMS architecture. For higher resolution sensors, future work would be needed to overcome the memory limitations present in [2] if the same high throughput performance is to be achieved by the local-flow implementation. Regardless, it has been demonstrated that the hARMS architecture provides a configurable solution for the real-time computation of aperture-robust event-based optical flow, enabling the use of optical flow for higher level event-based vision on embedded platforms.

Leveraging the fARMS and hARMS algorithms for fast computation of true flow, this research also presents a novel flow-based event-stream compression method. This FBC method relies on event prediction using flow estimates in order to reconstruct the event stream while only sending a limited number of the original events. The FBC method could also be paired with other existing methods of event compression, allowing for further increases in the achieved compression ratio. In this research the effect of the timing parameters on the evaluation metrics of CR, ER, and distance was evaluated and showed that a compression ratio >2.0 can be consistently achieved while maintaining a low distance between the reconstructed event stream and the original uncompressed event stream. These results demonstrate that the fast optical flow calculation provided by the fARMS and hARMS algorithm can be effectively used for event-stream compression, allowing for efficient communication of event data over low bandwidth mediums, while maintaining high reconstruction quality for input into the end-use application.

6.0 Future Work

The field of neuromorphic computing and sensing is rapidly evolving and presents promising areas of future research. The high-speed true-flow calculation enabled by the hARMS architecture means that the local flow computation will become the bottleneck as sensor sizes increase. Local-flow algorithms, while fast, would still benefit from optimization that removes the dependence on a frame of events. Future research could leverage the optimization techniques used in the fARMS algorithm to develop a fully neuromorphic computation pipeline.

The compression method introduced in this research shows that event-based optical flow can be used to further compress the event stream beyond the inherent compression performed by the event-based vision sensor. Future work on this compression will focus on adaptive selection of system timing parameters and more complex selection of events to transmit. As scene dynamics change, the optimal timing parameters change. Dynamically setting these parameters based on factors like average flow speed and event rate could improve overall reconstruction quality. Furthermore, the development of more complex transmission rules could allow more optimal selection of which events to transmit. Finally, the application of other compression methods to the output of the FBC method proposed will be investigated to enable higher end-to-end compression ratios by leveraging the event reduction achieved by flow-based compression.

Bibliography

[1] Himanshu Akolkar, Sio Hoi Ieng, and Ryad Benosman. Real-time high speed motion prediction using fast aperture-robust event-driven visual flow. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[2] Myo Tun Aung, Rodney Teo, and Garrick Orchard. Event-based Plane-fitting Optical Flow for Dynamic Vision Sensors in FPGA. In *Proceedings - IEEE International Symposium on Circuits and Systems*, volume 2018-May, 2018.

[3] Srutarshi Banerjee, Zihao W Wang, Henry H Chopp, Oliver Cossairt, and Aggelos K Katsaggelos. Lossy event compression based on image-derived quad trees and poisson disk sampling. In *2021 IEEE International Conference on Image Processing (ICIP)*, pages 2154–2158. IEEE, 2021.

[4] Ryad Benosman, Charles Clercq, Xavier Lagorce, Sio Hoi Ieng, and Chiara Bartolozzi. Event-based visual flow. *IEEE Transactions on Neural Networks and Learning Systems*, 25(2), 2014.

[5] Ryad Benosman, Sio Hoi Ieng, Charles Clercq, Chiara Bartolozzi, and Mandyam Srinivasan. Asynchronous frameless event-based optical flow. *Neural Networks*, 27, 2012.

[6] Zhichao Bi, Siwei Dong, Yonghong Tian, and Tiejun Huang. Spike coding for dynamic vision sensors. In *2018 Data Compression Conference*, pages 117–126. IEEE, 2018.

[7] Davis Blalock, Samuel Madden, and John Guttag. Sprintz: Time series compression for the internet of things. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(3):1–23, 2018.

[8] Kwabena A. Boahen. Point-to-point connectivity between neuromorphic chips using address events. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 47(5), 2000.

[9] Christian Brandli, Raphael Berner, Minhao Yang, Shih Chii Liu, and Tobi Delbruck. A 240 $\times$ 180 130 dB 3 μs latency global shutter spatiotemporal vision sensor. *IEEE Journal of Solid-State Circuits*, 49(10), 2014.

[10] Siwei Dong, Zhichao Bi, Yonghong Tian, and Tiejun Huang. Spike coding for dynamic vision sensor in intelligent driving. *IEEE Internet of Things Journal*, 6(1):60–71, 2018.

[11] Thomas Finateu, Atsumi Niwa, Daniel Matolin, Koya Tsuchimoto, Andrea Mascheroni, Etienne Reynaud, Pooria Mostafalu, Frederick Brady, Ludovic Chotard, Florian Legoff, Hirotsugu Takahashi, Hayato Wakabayashi, Yusuke Oike, and Christoph Posch. 0 A 1280×720 Back-Illuminated Stacked Temporal Contrast Event-Based Vision Sensor with 4.86μm Pixels, 1.066GEPS Readout, Programmable Event-Rate Controller and Compressive Data-Formatting Pipeline. In *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, volume 2020-February, 2020.

[12] Andrew C Freeman and Ketan Mayer-Patel. Lossy compression for integrating event cameras. In *2021 Data Compression Conference (DCC)*, pages 53–62. IEEE, 2021.

[13] Guillermo Gallego, Tobi Delbruck, Garrick Michael Orchard, Chiara Bartolozzi, Brian Taba, Andrea Censi, Stefan Leutenegger, Andrew Davison, Jorg Conradt, Kostas Daniilidis, and Davide Scaramuzza. Event-based Vision: A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[14] Massimiliano Giulioni, Xavier Lagorce, Francesco Galluppi, and Ryad B. Benosman. Event-based computation of motion flow on a neuromorphic analog neural platform. *Frontiers in Neuroscience*, 10:35, 2016.

[15] Berthold K.P. Horn and Brian G. Schunck. Determining optical flow. *Artificial Intelligence*, 17(1-3):185–203, 8 1981.

[16] Nabeel Khan, Khurram Iqbal, and Maria G Martini. Lossless compression of data from static and mobile dynamic vision sensors-performance and trade-offs. *IEEE Access*, 8:103149–103163, 2020.

[17] Nabeel Khan, Khurram Iqbal, and Maria G Martini. Time-aggregation-based lossless video encoding for neuromorphic vision sensor data. *IEEE Internet of Things Journal*, 8(1):596–609, 2020.

[18] Jianing Li, Yihua Fu, Siwei Dong, Zhaofei Yu, Tiejun Huang, and Yonghong Tian. Asynchronous spatiotemporal spike metric for event cameras. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[19] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. A 128 × 128 120 dB 15 μs latency asynchronous temporal contrast vision sensor. *IEEE Journal of Solid-State Circuits*, 43(2), 2008.

[20] Min Liu and Tobi Delbruck. Block-matching optical flow for dynamic vision sensors: Algorithm and FPGA implementation. In *Proceedings - IEEE International Symposium on Circuits and Systems*, 2017.

[21] Min Liu and Tobi Delbruck. Adaptive time-slice block-matching optical flow algorithm for dynamic vision sensors. In *British Machine Vision Conference 2018, BMVC 2018*, 2019.

[22] Bruce D Lucas, Takeo Kanade, et al. An iterative image registration technique with an application to stereo vision. Vancouver, British Columbia, 1981.

[23] Elias Mueggler, Henri Rebecq, Guillermo Gallego, Tobi Delbruck, and Davide Scaramuzza. The event-camera dataset and simulator: Event-based data for pose estimation, visual odometry, and SLAM. *International Journal of Robotics Research*, 36(2), 2017.

[24] Christoph Posch, Daniel Matolin, and Rainer Wohlgenannt. A QVGA 143 dB dynamic range frame-free PWM image sensor with lossless pixel-level video compression and time-domain CDS. In *IEEE Journal of Solid-State Circuits*, volume 46, 2011.

[25] Bodo Rueckauer and Tobi Delbruck. Evaluation of event-based algorithms for optical flow with ground-truth from inertial measurement sensor. *Frontiers in Neuroscience*, 10(APR), 2016.

[26] Bongki Son, Yunjae Suh, Sungho Kim, Heejae Jung, Jun Seok Kim, Changwoo Shin, Keunju Park, Kyoobin Lee, Jinman Park, Jooyeon Woo, Yohan Roh, Hyunku Lee, Yibing Wang, Ilia Ovsiannikov, and Hyunsurk Ryu. A 640×480 dynamic vision sensor with a 9µm pixel and 300Meps address-event representation. In *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, volume 60, 2017.

[27] Xilinx. *Zynq-7000 SoC Data Sheet: Overview*, 7 2018. Rev. 1.11.1.

[28] Alex Zhu, Liangzhe Yuan, Kenneth Chaney, and Kostas Daniilidis. EV-FlowNet: Self-Supervised Optical Flow Estimation for Event-based Cameras. In *Robotics: Science and Systems XIV*. Robotics: Science and Systems Foundation, 6 2018.

[29] Alex Zihao Zhu, Dinesh Thakur, Tolga Özaslan, Bernd Pfrommer, Vijay Kumar, and Kostas Daniilidis. The Multivehicle Stereo Event Camera Dataset: An Event Camera Dataset for 3D Perception. *IEEE Robotics and Automation Letters*, 3(3), 2018.