ParaHist: FPGA Implementation of Parallel Event-Based Histogram for Optical Flow Calculation

Mohammad Pivezhandi, Phillip H. Jones, Joseph Zambreno
Electrical and Computer Engineering, Iowa State University
Ames, Iowa 50011
{*mpvzhndi, phjones, zambreno*}*@iastate.edu*

***Abstract*—In this paper, we present an FPGA-based architecture for histogram generation to support event-based camera optical flow calculation. Our proposed histogram generation mechanism reduces memory and logic resources by storing the time difference between consecutive events, instead of the absolute time of each event. Additionally, we explore the trade-off between system resource usage and histogram accuracy as a function of the precision at which time is encoded. Our results show that across three event-based camera benchmarks we can reduce the encoding of time from 32 to 7 bits with a loss of only approximately 3% in histogram accuracy. In comparison to a software implementation, our architecture shows a significant speedup.**

***Keywords*-Event-based camera sensors; FPGA-based architecture; Histogram generation**

I. Introduction

Event-based cameras use biologically inspired sensors, which only transmit information of pixels that have changed, to overcome throughput limitations of traditional frame-based cameras. This behavior has the potential to allow event-based cameras to achieve much lower latency, and greater effective frame update rates than frame-based cameras. This has made them attractive for use in applications such as blurred video reconstruction, high-speed object tracking, and real-time collision avoidance for autonomous vehicles or unmanned air vehicles [1], [2]. However, to fully realize the potential of event-based cameras, the architecture of a computer vision system must address new challenges related to both processing asynchronous sparse events and managing meta-data associated with each pixel.

In this paper, we present an FPGA-based architecture for histogram generation to support event-based camera optical flow. A key feature of our design is the compression of time information. By storing the time difference between consecutive events, instead of the absolute time of each event, we significantly reduce memory and logic resources. This design further supports resource savings by flexibly allowing the trade-off between the precision at which time is recorded, and histogram accuracy. Our architecture can process over 200 million events/sec, and shows promise for being integrated as part of a larger event-based camera-driven computer vision system for optical flow or object tracking.

II. Architecture of Event-based Histogram Generator

Figure 1 illustrates our pipelined architecture for generating event-based histograms. In summary, data flows through the pipeline stages as follows: 1) the Event Mapping stage streams events into an input FIFO, compresses, and then stores them into a two dimensional array of Block RAMs, 2) the Noise Removal stage removes events that are outside of a threshold, 3) the Timestamp Decompression stage decompresses events to remove outliers, 4) the Histogram Update stage updates the histogram based on the decompressed events, 5) the Write Back stage updates the previous timestamp of the compressed events in the Event Mapping stage with the current timestamp from the Histogram Update stage, and 6) the updated histograms are placed into FIFOs for an application to use for gradient calculations.

This section next provides details for each of the six stages of the architecture.

A. Event Memory Mapping

This stage of the pipeline assumes that events are streamed from an event-based camera into an input FIFO formatted using an address-event representation (AER). AEDAT 2.0 is the specific AER format assumed in this paper, shown in Fig. 2. These events are then read from the FIFO, are compressed, and are then stored in an array of RAMs. Our compression approach concatenates consecutive timestamps for a pixel, allowing us to store the difference between timestamps, instead of the absolute time for each event. For the case where we maintain full time precision, the size of timestamps

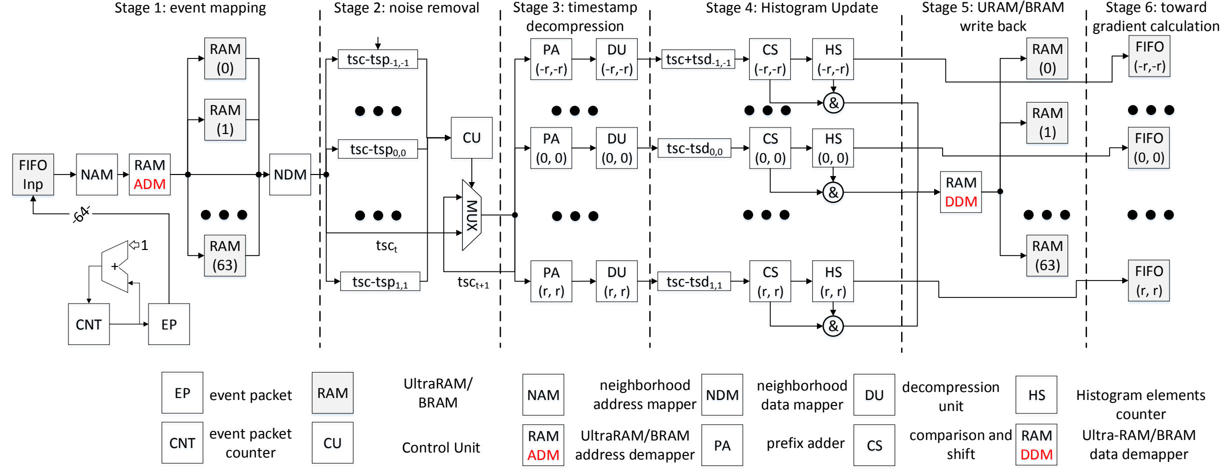

Figure 1. Event based Histogram Computation Unit

reduces from 32 bits to 18 bits, when moving from storing absolute time to time differences.

Figure 3 illustrates our compressed event format, where multiplied events are concatenated into a ring buffer structure. The size of each ring buffer structure is W_{ts}, as computed using Eqn. 1, where W_{data} is the bits of precision required, W_{dt} is the number of bits to store time differences, H_s is the histogram capacity, and PN is the data precision.

$$W_{ts} = W_{data} \times PN - (H_s * W_{dt}) - \lfloor log(H_s) \rfloor \quad (1)$$

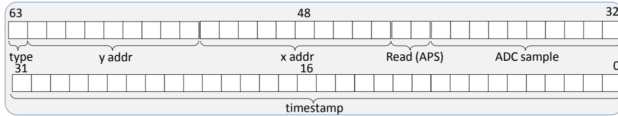

Figure 2. Data representation for an event in AEDAT 2.0. Each event comprises of 32 bits for an events timestamp, and 32 bits consisting of the polarity (type), address (xaddr and yaddr), pixel intensity (ADC), and a select part (Read APS) for IMU, APS and signal level options.

The compressed data is stored into the array of RAMs using a 2-D addressing scheme. The lowest 3 bits of the x address, and lowest 3 bits of the y address (see Fig. 2) are used to index into this array of RAMs. This addressing scheme stores neighboring pixels regions into separate RAMs, thus allowing for parallel access to up to 64 pixel regions at a time (i.e. an 8x8 grid of pixel regions).

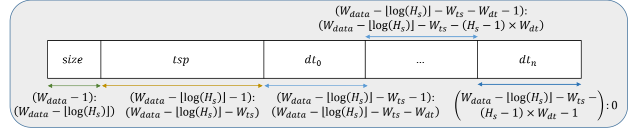

Figure 3. Data representation for each line of memory unit

B. Noise removal

In this stage we use the concept of a background activity filter [3] for noise removal. When there is no noise, a moving object will cause events to occur temporally close in time in the spatial neighborhood of an object. However in a dynamic scene with noise, an event within an object's spatial neighborhood that occurs temporally far away from the last event is likely cause by a random glitch, which could come from a number of sources. We use a time threshold of (t_n) to identify events that are too far away temporally as noise events, and remove them. Equation 2 shows how this threshold is used, where tsc refers to the current timestamp and $tsp_{1:1;\ 1:1}$ are previous timestamps within neighboring regions.

$$tsc - tsp_{1:1;\ 1:1} < t_n \quad (2)$$

C. Timestamp Decompression

This stage is responsible for decompressing timestamps, and removing outliers. Each stage of this process is illustrated in Fig. 4. First a parallelized prefix adder is used to sum the time differences concurrently. We implement the prefix adder using the radix4-Sklansky

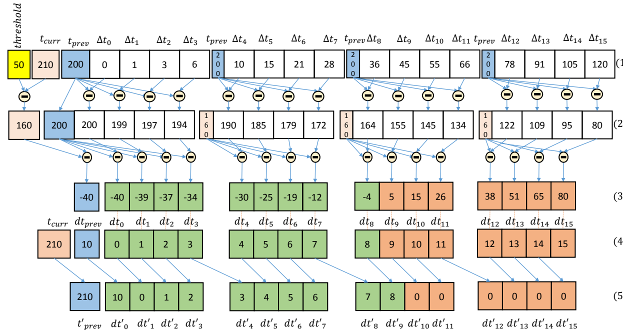

Figure 4. Decompressing and comparing timestamps for the histogram update stage. 1) the result from the prefix adder, 2) the decompressed data after subtracting the previous timestamp. 3) subtracting the previous timestamp from the current timestamp for outlier detection. 4) positive values mean timestamps within the threshold value. 5) one right shifting the ring buffer elements to store the recent timestamp in place of the previous timestamp.

approach [4], which makes efficient use of hardware resources. After summing time differences, we subtract these from the previous timestamp to obtain decompressed timestamps (line (2)). Outliers are removed if they are greater than the current time minus the threshold (i.e. $t_{curr} - threshold$). In Fig. 4, the current time is $t_{curr} = 210$, the threshold is $threshold = 50$, and their difference is 160. Thus, in line (3) of Figure 4 positive times are removed by zeroing them out in line (5). Additionally, it can be seen when moving from line (4) to line (5) that the timestamps are moved to the right by one in the ring buffer. Note that though this method increases the number of operators from 26 to 36 when compared to Brent-Kung [5], it decreases the number of steps from $2log_2(n)$ to $log_2(n)$.

D. Histogram Update

After decompressing the timestamps and removing outliers, the remaining values (shown in green in line (4) of Fig. 4) are used to update the histogram. This process simply involves counting the green boxes to increment the appropriate bins of the histogram, and shifting the event ring buffer by one to the right.

III. Results

Though event-based camera sensors are inherently asynchronous, multiple pixels can spike simultaneously. This asynchronous behavior makes sub-microsecond processing time in FPGAs viable when sufficient parallelism is applied. We used two separate input benchmarks for our evaluation: 1) the rotating disk benchmark is a disk with eight compartments in which the camera moves around its z dimension. 2) The translating sinusoid benchmark is a pattern that varies sinusoidally in a horizontal direction, with the camera is panned clockwise around the y-axis. This results in a model that shifts to the left. Since the per-frame latency equivalent in event-based cameras is on the order of a single microsecond, for this benchmark, that has as many as 1396 pixels spiking at the same time in translating boxes, implies that our hardware solution requires a processing time of approximately of 10^{6}=1396 or about 0:71 nano-second to keep up with our inputs.

To measure the resource usage of our proposed architecture, we synthesized various configurations of our VHDL design using Vivado HLx 2018.1. Hardware utilization is presented in Table I in which several trends can be observed. First, the resource consumption relates to the data width of each line of memory, and each of these lines represents a ring buffer with variable element width as shown in Fig. 3. To meet the fixed configuration of Ultra-RAM (RAM_U) modules, the width of ring buffer in each line of memory would be 72 bits multiplicands of precision. When the value of precision is more than one, the data width exceeds 72 bits, which results in a hybrid set of BRAMs (RAM_B) and RAM_U modules. The current timestamp data width should be similar to the previous timestamp in width, and, according to Eqn. 1, the final data width depends on the capacity of the histogram elements counter, delta-times, and a precision factor. For example, with a histogram capacity of 4 bits, delta times width of 4 bits, and a precision factor of 1, we would have $72 - (4 + 16 \times 4) = 4$ bits for each timestamp. Moreover, the required resources on the FPGA relates to the search distance. This search distance (radius) value exponentially increases the number of parallel regions for processing neighboring pixels, i.e., horizontal elements in Fig. 1. In Table I, resource consumption is based on a design space exploration with a data-width of 72 bits with precision, multiplicands of 1, 2, 3, and 4, and search regions of 3*3, 5*5, and 7*7.

For throughput and processing time analysis, the target frequency is the key factor. We divide the throughput analysis into on-chip and off-chip considerations. The On-chip throughput is equal to the $Frequency \times DataWidth \times SearchDistance$. As presented in Fig. 5, in a constant frequency, increasing the search distance and data width increases the on-chip throughput from about 100 Gbps to 2700 Gbps. On the other hand, the off-chip throughput is equal to the target frequency, and this design has an off-chip throughput of more than 200 M event/sec. On