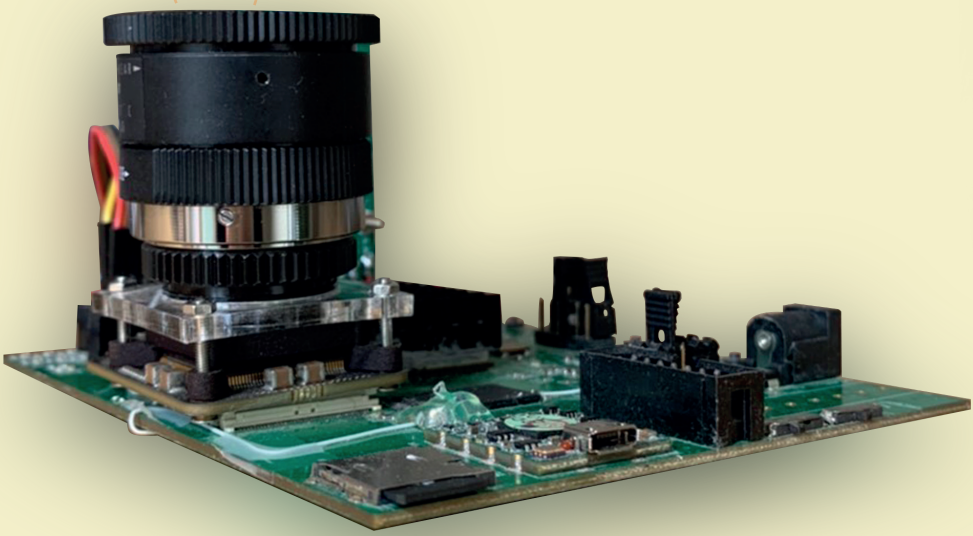
Min Liu

HARDWARE OPTICAL FLOW FROM EVENT CAMERAS



DISS. ETH NO. 27817

DISS. ETH NO. 27817

HARDWARE OPTICAL FLOW FROM EVENT CAMERAS

A dissertation submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

MIN LIU

born on 27 May 1991
citizen of China

accepted on the recommendation of

Prof. Dr. Tobi Delbruck, examiner
Prof. Dr. Shih-Chii Liu, co-examiner
Prof. Dr. Benjamin Grewe, co-examiner
Dr. Garrick Orchard, co-examiner

2021

Dedicated to my family

ABSTRACT

Electronic cameras are now ubiquitous throughout daily life. Mobile robotics has adopted cameras to provide environment perception. However, it is still challenging to solve essential problems such as Visual Odometry (**VOD**) and Simultaneous Localization And Mapping (**SLAM**) with traditional cameras under poor lighting conditions or for high speed motion.

A new type of camera that that works differently from the conventional frame-based camera has become more widely popular for its sparse output, micro-second temporal resolution, and high dynamic range. This camera is called 'event camera' or Dynamic Vision Sensor (**DVS**) because it is activity-driven instead of being frame-based. The time resolution of DVS is on the order of a few microseonds under good lighting conditions, which is about $10,000$ times faster than a smartphone camera with a sample rate of 120Hz. This characteristic makes it suitable for high-speed applications such as mobile robotics or drones.

Optical Flow (**OF**) is a low-level but essential problem in the computer vision community. It is a fundamental module for most VOD or SLAM systems. Nevertheless, frame-based optical flow usually suffers from motion blur in the presence of fast motion or extreme image conditions such as under/over exposure. By benefiting from the super high temporal resolution, event-based OF provides more possibilities to solve these problems. Event-based OF is thus attracting growing attention. However, most of them have primarily focused on the accuracy from the software side and overlooked the real-time performance from the hardware side. Hence, in this thesis, we present the first hardware optical flow estimator along with a hardware corner detector for event-based cameras.

First, we propose a novel event-based optical flow estimation method called Adaptive Block Matching Optical Flow (**ABMOF**). Events are accumulated in rotating event slices using the area event count slice rotation method, that adapts to scenes with various levels of spatial sparsity. The slice rotation event count number is feedback-controlled by the average optical flow matching distance. It use a Coarse to Fine (**CTF**) spatial architecture. The multiscale Block Matching (**BM**) size is 25×25 pixels, and the flow vectors span up to a 30-pixel match distance.

Second, we designed a hardware corner detection method called Slice-based FAST (**SFAST**). SFAST is adapted from a popular frame-based cor-

ner detection method Features from Accelerated Segment Test (**FAST**). Corners are detected by streaks of accumulated events on event slice rings of radius 1 and 2 pixels. Compared with another event-based variant of FAST called Event-Based time surface FAST (**EFAST**), SFAST is more accurate, more compatible with ABMOF and more efficient.

Third, we designed a new powerful event camera platform based on Xilinx's Zynq SoC 7100. This System On Chip (**SOC**) features a dual-core ARM Cortex A9 microcontroller mated with a Xilinx Virtex family Field Programmable Gate Array (**FPGA**). The platform is called `Davis346Zynq`. The SOC is directly connected to the DVS sensor. The platform has 512MB DDR3 and 1GB Flash memory. It supports three power supply methods: external 5V plug-in, external Universal Serial Bus (**USB**), and LiPo battery. The interfaces to the platform consist of one customized USB2.0 and one Video Graphics Adaptor (**VGA**), two UARTs, one I^2C, one MicroSD, and several GPIO extensions.

Finally, SFAST and the ABMOF are integrated and verified on the `Davis346Zynq`. The combination of these two Intellectual Propertys (**IP**) is called Event-driven Optical Flow (**EDFLOW**). Optical flow vectors are measured only at the corners and can be computed in 1 us at 100 MHz clock frequency. The FPGA processes the Sum of Absolute Differences (**SAD**) BM at 123 GOp/s, the equivalent of 1230 Op/clock cycle. EDFLOW is 8 times more accurate than the previous best DVS FPGA optical flow implementation on the *slider_hdr_far* scene from [1]. Compared with Convolutional Neural Network (**CNN**)-based optical flow, EDFLOW is less accurate. However, it burns about 100 times less power and is 10 times quicker.

ZUSAMMENFASSUNG

Elektronische Kameras sind im täglichen Leben allgegenwärtig. In der mobilen Robotik werden Kameras für die Umgebungswahrnehmung verwendet. Es ist jedoch immer noch eine Herausforderung, wichtige Berechnungen wie VOD und SLAM mit herkömmlichen Kameras, besonders bei bei schlechten Lichtverhältnissen oder bei Hochgeschwindigkeitsbewegungen, zu bewerkstelligen.

Ein neuer Kameratyp, der sich von herkömmlichen einzelbildbasierten Kameras unterscheidet, ist aufgrund dünnbesetzter Ausgabe, der zeitlichen Auflösung im Mikrosekundenbereich und des weiten Dynamikumfangs immer beliebter geworden. Diese Kamera wird 'Ereigniskamera' oder DVS genannt, weil sie aktivitätsgesteuert und nicht bildbasiert ist. Die Zeitauflösung der DVS liegt bei guten Lichtverhältnissen in der Grössenordnung von wenigen Mikrosekunden, was etwa 10'000 mal schneller ist als eine Smartphonekamera mit einer Abtastrate von 120Hz. Diese Eigenschaft macht sie geeignet für Hochgeschwindigkeitsanwendungen wie mobile Robotik und Drohnen.

Optischer Fluss / OF ist ein wichtiges Thema und Forschungsgebiet von Computer Vision. Des Weiteren ist es eine Grundvoraussetzung für die meisten VOD- oder SLAM-Systeme. Der bildbasierte optische Fluss leidet normalerweise unter sehr schnellen Bewegungen oder extremen Bildbedingungen wie Unter-/Überbelichtung. Durch die extrem hohe zeitliche Auflösung bietet ereignisbasiertes OF mehr Möglichkeiten, diese Probleme zu lösen. Das ereignisbasierte OF findet daher zunehmend Beachtung, die bestehenden Ansätze fokussieren sich jedoch hauptsächlich auf die Genauigkeit auf der Softwareseite und ignorieren die Echtzeitleistung und Geschwindigkeit auf der Hardwareseite. Daher präsentieren wir in dieser Arbeit den ersten Hardware-Optical-Flow-Schätzer zusammen mit einem Hardware-Ecken- detektor für ereignisbasierte Kameras.

Als erstes schlagen wir ein neuartiges ereignisbasiertes Verfahren zur Berechnung des optischen Flusses vor und nennen es ABMOF. Ereignisse werden in rotierenden Ereignisabschnitten akkumuliert, wobei die Methode der rotierenden Ereigniszählung pro Fläche verwendet wird, die sich an Szenen mit unterschiedlichen Graden an räumlicher Dichte anpasst. Die Anzahl der Ereignisse in den rotierenden Abschnitten wird gesteuert

durch Rückkoppelung der durchschnittlichen Distanz der Punkte imoptischen Fluss. Das Verfahren verwendet eine CTF Raumstruktur.

Die Grösse des multiskalen BM beträgt 25 × 25 Pixel und die übereinstimmenden Flussvektoren erstrecken sich über eine Entfernung von 30-Pixel.

Zweitens haben wir eine Hardware-Eckenerkennungsmethode namens SFAST entwickelt. SFAST wurde von einer beliebten Einzelbild-Eckenerkennungsmethode genannt FAST übernommen. Ecken werden durch Streifen von akkumulierten Ereignissen auf Ereignisabschnittsringen mit Radius 1 und 2 Pixel erkannt. Im Vergleich zu einer anderen ereignisbasierten Variante von FAST namens EFAST ist SFAST genauer, kompatibel mit ABMOF und effizienter.

Drittens haben wir eine neue leistungsstarke Eventkameraplattform entwickelt, die auf dem Zynq-SoC von Xilinx 7100 basiert. Dieses SoC verfügt über einen Dual-Core ARM Cortex A9 Mikrocontroller, der mit einer Xilinx Virtex Familie FPGA kombiniert ist. Die Plattform heisst Davis346Zynq. Der SoC ist direkt mit dem DVS-Sensor verbunden. Die Plattform hat 512MB DDR3 und 1GB Flash-Speicher. Sie unterstützt drei Stromversorgungsmethoden: Externes 5V, externes USB, sowie LiPo-Akku. Die Schnittstellen zur Plattform bestehen aus einem angepassten USB2.0, VGA, zwei UARTs, einem I^2C, einer MicroSD und mehreren GPIO-Erweiterungen.

Zum Abschluss werden SFAST und ABMOF auf dem Davis346Zynq integriert und verifiziert. Die Kombination dieser beiden Erfindungen / IP heisst EDFLOW. Optische Flussvektoren werden nur an den Ecken gemessen und können in 1 us bei 100 MHz Taktfrequenz berechnet werden. Der FPGA verarbeitet den SAD BM mit 123 GOp/s, das entspricht 1230 Op pro Taktzyklus. EDFLOW ist 8 mal genauer als die bisherige beste DVS FPGA OF Implementierung in der *slider_hdr_far* Szene aus [1]. Im Vergleich zum CNN basierten optischen Fluss ist EDFLOW weniger genau. Es benötigt jedoch etwa 100 mal weniger Strom und ist 10 mal schneller.

ACKNOWLEDGEMENTS

The PhD career is a specific and unforgettable life journey for me. Looking back on the past years, I am fortunate that so many people are standing and supporting behind me all the time. Without them, this PhD thesis could not be finished.

First of all, I want to thank is my supervisor Prof. Tobi Delbruck. Tobi provides me not only financial support but also research support. Whenever I encountered some problems, he was there and was always ready to provide me some solutions. I learned a lot from him. He seems passionate about everything, from circuits to algorithms, from computer mice to slasher, from Dextra to Trixsy. He is conscientious about every detail. When we wrote papers together, he checked the text word by word. Without his guidance and constant feedback, I cannot finish my PhD project. It is hard to describe how much I appreciate his help. I want to express my best gratitude to him.

At the same time, I want to thank Prof. Shih-Chii Liu. Although we did not work on one project together, she gave me many suggestions not only on the research side but also on the life side. Shih-Chii and Tobi sometimes held parties and invited us to their home. Those are also lovely moments. Many thanks to her for being my co-examiner and giving a lot of feedback on my thesis.

I owe special thanks to Prof. Benjamin Grewe and Dr. Garrick Orchard, who kindly agreed to be my co-examiners and send me a lot of comments on my thesis.

I want to thank Chang Gao for the excellent discussions and suggestions on Vivado HLS. I want to thank as well Yuhuang Hu for helping me make the case for my hardware camera. I would also like to show my gratitude to Luca Longinotti and Dr. Chenghan Li from iniVation for helping me debug on the DAVIS sensor. Many thanks go to Joachim Ott for proofreading of the abstract in German. I want to thank my colleagues from my group, the best group, Sensors research group, and Institute of Neuroinformatics. In particular, Dr. Stefan Braun who invited me to his sweet wedding and taught me some German words, Dr. Hongjie Liu who kindly sent me a gift card on my birthday, Dr. Dongchen Liang, Sunil Sheelavant, Shu Wang, Xi Chen, Yingqiang Gao and Zhenming Yu. It is really lucky to be here and meet all of them.

I want to thank all my friends who support me. I wish particularly to thank my friend Cheng-Ing Wu for encouraging me and giving me support during the rock bottom of my PhD life. I also want to thank Wanting Chiu, who is standing on my side all the time and help me embellish some figures in the thesis, especially her help on the cover design of this thesis.

Finally, I would like to thank my family. We are a normal four-people family in China. They worked very hard to support my brother and me in our studies. My parents give all their love to us. Home is always a haven for the soul. Whatever I meet in life, good or bad, happy or sad, they are always ready to share with me or comfort me. They are the people I can always count on. Without them, I would not be here. They were, are, and will always be in my heart. I love them forever.

I gratefully acknowledge financial support from the Swiss National Center of Competence in Research Robotics (NCCR Robotics).

CONTENTS

ACRONYMS

AAE Average Angular Error
ABMOF Adaptive Block Matching Optical Flow
AEE Average Endpoint Error
AER Address Event Protocol
APS Active Pixel Sensor
ASIC Application Specific Integrated Circuit
BHD Binary Hamming Distance
BM Block Matching
BMOF Block Matching Optical Flow
BRAM Multipurpose Block RAM memory module in FPGA
CCD Charge-coupled Device
CMOS Complementary metal–oxide–semiconductor
CNN Convolutional Neural Network
CPI Count Per Inch
CPLD Complex Programmable Logic Device
CRC Cyclic redundancy check
CTF Coarse to Fine hierarchical search strategy used in BMOF
CV Computer Vision
DAVIS Dynamic and Active pixel Vision Sensor
DB Daughter Board
DDR DDR memory unit that transfers data on both clock edges
DMA Direct Memory Access
DNN Deep Neural Network
DRAM Dynamic RAM
DS Diamond search method
DS Direction Selective model of motion detection in biological vision, usually either Hassenstein-Reichhardt or Barlow-Levick type
DSP Digital Signal Processing unit
DVS Dynamic Vision Sensor
EBLK Event-based Lucas-Kanade
ED Event Density
EDA Electronics Design Automation
EDFLOW Event-driven Optical Flow
EFAST Event-Based time surface FAST
FAST Features from Accelerated Segment Test

FIFO First In First Out memory
FPGA Field Programmable Gate Array
FPS Frame Per Second
FSM Finite State Machine
GF Global Flow
GPU Graphics Processing Unit
GT Ground Truth
HD Hamming Distance
HDL Hardware Description Language
HLS High Level Synthesis
IAS Image Acquisition Sensor
IC Integrated Circuit
II Initial Interval
IMU Inertial Measurement Unit
IP Intellectual Property
JTAG Joint Test Action Group
LK Lucas-Kanade
LP DVS OF method that fits a plane to local event cloud
LSB Least Significant Bit
LUT LookUp Table
MAV Micro Aerial Vehicles
MB Mother Board
MCU Microcontroller Unit
ME Motion Estimation
MSB Most Significant Bit
NPC Number of Parallel Computation
NRZI Non-Return-to-Zero Inverted Code
OF Optical Flow
OPS Operations Per Second
PCB Printed Circuit Board
PE Processing Elements
PID Packet Identifier
PL Programmable Logic
PLL Phase-locked loops
PRM Pixel Rendering Module
PS Processing System
RANSAC RANdom SAmple Consensus
RB A Reference Block centered on the event location in the t-d1 slice
ROS Robot Operating System

SA Search Area for block matching
SAD Sum of Absolute Differences
SAE Surface of Active Events
SD Secure Digital
SDK Software Development Kit
SDSoC Software-Defined System On Chip
SFAST Slice-based FAST that uses accumulated event count slices for detecting keypoints
SIFT Scale Invariant Feature Transform
SILC Speed Invariant Learned Corners
SITS Speed Invariant Time Surface
SLAM Simultaneous Localization And Mapping
SNN Spiking Neural Network
SOC System On Chip
SRAM Static RAM
SUSAN Smallest Univalue Segment Assimilating Nucleus)
TB A candidate or best-match Target Block for block matching in the t-d2 slice
TI Timestamp Image; image of latest event timstamps, same as Surface of Active Events
ULPI UTMI+Low Pin Interface
USB Universal Serial Bus
UTMI USB Transceiver Macrocell Interface
VGA Video Graphics Adaptor
VIO Visual-Inertial Odometry
VOD Visual Odometry

1

INTRODUCTION

"When we walk or drive or even move our heads, our view of the world changes. Even when we are at rest, the world around us may not be; objects fall, trees sway, and children run. Motion of this sort, and our understanding of it, seems so straightforward that we often take it for granted. In fact, this ability to understand a changing world is essential to survival; without it, there would be no continuity to our perceptions."

— Michael Black (1992, Ph.D. thesis)

1.1 FROM ZOOPRAXISCOPE TO EVENT CAMERA

1.1.1 *The man who stopped time*

Film or motion picture is quite familiar to us in modern days. However, the history of the film actually started from a bet. In 1872, a racehorse owner, who also served as the former governor of California, Leland Stanford[1] asked a question to the public, if a horse could "fly" in a gallop. He bet on the side that advocates "unsupported transit," which means that all four hooves are in the air while it is running. To solve this bet, he invited the top photographer Eadweard Muybridge. It might be a simple question in current days. However, limited by the camera technology at that time, which requires fifteen seconds to even several minutes for exposure, it is impossible to capture the motion generated by a horse running at about 40 feet per second. The first thing Muybridge did was to speed up the exposure time. He managed to design a new method and decreased the exposure time to a fraction of one second. Then, he placed 12 cameras in parallel with a horse's race path, and a wire controlled each camera's shutter. While the horse was running, the horse's chest would break the wire and thus trigger the shuttle. The whole experiment was done in June of 1878 beside the track on the Palo Alto Stock Farm. The result of the experiment is 12 successive photos. The successive photos proved Stanford's idea, which was

1 He is also the founder of the famous Stanford University.

different from the scene that people saw in typical paintings. In contrast to the four legs that were extended in the paintings of the day, the photos showed that they gathered under the horse. For the first time, it captures ephemeral details that the human eye cannot distinguish at such speeds, such as the position of the leg or the angle of the tail. Muybridge quickly

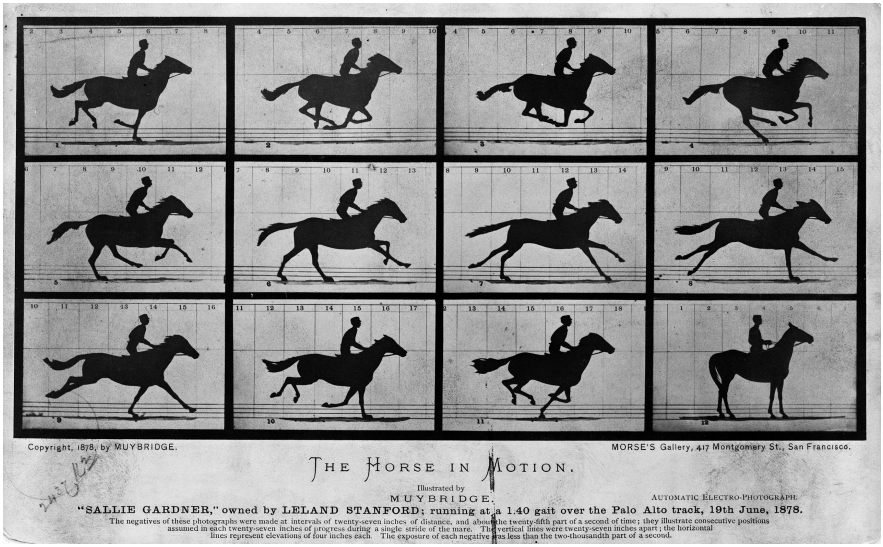


FIGURE 1.1: Card with "Sallie Gardner" in an altered 1879 edition. Sallie Gardner is the race horse's name. The 12 successive photos taken by Muybridge are made as cards and published by Morse's gallery. These cards show the details of the gallop of a horse. It is very clear from these cards that all four legs are in the air while the horse is running. Figure courtesy of Library of Congress Prints and Photographs Division.

began selling his photocopies as "photo cards." Morse's gallery publishes twelve cards created based on Muybridge's photos. These cards are shown in 1.1.

To make the horse motion clearer, Muybridge invented a new machine to project moving photographic photos. This machine rotates at a specific speed and shows the successive photos one by one. When observed by human eyes, it looks like a "moving" horse. Nowadays, we know that the persistence of vision causes this phenomenon. Muybridge called this new machine zoopraxiscope, and it started the new era of film or the history of cinematography.

It might not be seen as a remarkable achievement in modern days. However, Muybridge was the first person who really "stopped" time. It was the first time that humans could cut fast motion into static motion pictures. Curiosity about high/super high-speed motion did not stop there. Faster and faster video cameras are still developed to capture short sequences at up to billions of frames per second.

1.1.2 *Modern image sensors*

Several years after the bet, the founder of the Kodak company, George Eastman, made the film camera almost ubiquitous in the world by improving the technology and making cameras smaller and smaller.

This section mainly reviews the digital camera. The time comes to 1969. Willard Sterling Boyle and George Elwood Smith from Nokia Bell Labs invented the Charge-coupled Device (**CCD**). The original purpose of the CCD was for memory devices. However, it is successfully used in the digital image sensor and becomes quite popular soon. CCDs transfer the charges triggered by the light to its neighbor, and this transfer is performed one by one until it arrives at the last capacitor. By repeating this process, the charge stored in each pixel is converted to a voltage. These voltages are sampled and stored in a memory device or sent to the other circuit for further processing. Kodak invented the first CCD camera in 1975. Since then, the camera history has entered into the digital era.

In 1993, NASA's Jet Propulsion Laboratory developed a Complementary metal–oxide–semiconductor (**CMOS**) Active Pixel Sensor (**APS**) sensor, simply called it APS in the following text. This sensor uses the CMOS technology which is also used for other digital ICs such as the microcontroller, which makes it possible to integrate the image sensor and the post-processing circuit such as the readout circuit, analogy to digital converter circuit, and even the memory circuit on one chip. Instead of only one amplifier for all pixels in CCD, APS cameras use an individual amplifier for each pixel. APS helps increase throughput while decreasing the cost and power compared with CCD. Once the fundamental invention of fully depleted photodiodes, correlated double sampling readout, and many other techniques enabled APS to start to match the extremely high quality of the existing CCD cameras, APS became the primary type of camera in the consumer electronics market.

1.1.2.1 *High-speed cameras*

Nowadays, image sensors are not only used to make specific professional cameras or videos; they are also embedded mainly into phones. It hence enhances the application of the images/videos. Image/video is not only used as a tool to record beautiful scenes, people, and precious moments. It is also used in such as facial recognition, surveillance, and even robotics. The sensor evolves in two directions to satisfy all kinds of requirements. One is to increase the resolution. Phone company Xiaomi published the first phone CC9, which includes a 108 million-pixel image sensor, in 2019. The other direction is to increase the Frame Per Second (**FPS**). FPS does not only require a fast camera, but also requires a fast Microcontroller Unit (**MCU**) for processing. iPhone 12 Pro, the best IOS phone released by Apple in 2020, could process 4K video at 60fps. This is enough for the most common recording and processing in real-time. However, it is still slow for high-speed or ultra-high-speed image/video processing applications such as flying drones, autonomous cars, and robotics.

TABLE 1.1 compares some of the state-of-the-art ultra-high-speed cameras. All of them are implemented by CMOS with a global shutter. Phantom could achieve the max FPS with a moderate resolution at the cost of around 80,000 dollars. Freefly Wave has the highest resolution and could run at a maximum of 422 FPS. Chronos 2.1-HD is the cheapest but has the lowest resolution. If we checked it from the real-time processing aspects, such as using them in robotics, Freely Wave would be a possible option since it is the lightest and consumes the least power.

1.1.3 *Dynamic vision sensors*

The sensors discussed in Sec.1.1.2 are called Frame-based vision sensors or cameras. Although some properties such as high temporal resolution could be comparably implemented in frame-based cameras as shown in TABLE 1.1, they are often bulky, power-insensitive and require cooling.

Let us go to check more details about TABLE 1.1. Cameras in TABLE 1.1 look provide some promising candidate cameras for high-speed applications, but actually none of them are good enough. The most common and significant problem for all of them is power consumption. Take Freely Wave as an example, it consumes the least power, but 24W is still far more away from that a mobile robotic which might be equipped with a battery could provide. The other problem is storage. Phantom T1340 reports that it could fill up the 144GB internal RAM in only 7.6 seconds when running at

Spec \ Model	Phantom T1340 [2]	Freefly Wave [3]	Chronos 2.1-HD [4]
Sensor type	Global shutter CMOS	Global shutter CMOS	Global shutter CMOS
Sensor size	27.6 mm x 26.3 mm	22.53 mm x 16.90 mm	19.2 mm x 10.8 mm
Resolution	2048 x 1952	4096 x 3072	1920x1080
FPS	3270	422	1000
Dynamic Range	61.4dB	66 dB	62.4 dB
Mechanical dimensions	203mm x 127mm x 127mm	150mm x 97mm x 47mm	155mm x 96mm x 67.3mm
Weight	4.5kg	0.716kg	1.06kg
Bit-depth	12	10	12
DC operation voltage	20-28V	12-26V	17-20V
Power	150W	24W	40W
Price	$80,000	$10,000	$5,000

TABLE 1.1: comparison between different ultra-high-speed cameras

the maximum FPS, 12bits, 2048x1952 resolution. Not to mention that more extensive storage consumes more power; it also makes the system more expensive and bulkier.

All frame-based vision sensors capture the images in a fixed interval and record them one by one. Eventually, it would record many redundant data, especially when the scene does not change too much. This redundant data occupies storage and consumes more power, which makes it not efficient. However, if we compare it with the visual processing in primates such as humans, we can find that human visual processing is powerful and effective. The human could finish many tasks which are considered “complicated" for the machine, such as object recognition, motion estimation, and classification easily and still maintain the brain at an approximate temperature. What is the mystery behind human eyes, and can we mimic

a silicon eye to solve the problems that current frame-based vision sensors are difficult or even impossible to solve.

DVS or called the event-based camera [5–10] was thus invented. It mimics human retina photoreceptors and retina ganglion cells. This new type of sensor is very different from the frame-based vision sensor as the human's retina neuro system works asynchronously. Therefore, the first difference is that DVS does not have a global clock that ticks the photodiodes to "precept" the world with a fixed rhythm.

Secondly, the output of DVS is quite different. For frame-based vision sensors, either CCD or APS, each pixel's voltage is triggered by the absolute brightness, and all pixels must be triggered within a small time window. However, DVS pixels independently react to brightness (log intensity) changes. If any pixel detects a brightness increase or decrease that exceeds a critical threshold amount relative to the previously memorized brightness, it generates an output spike and memorizes the new brightness value. It helps remove the static or redundant background and makes the camera more focused on the moving objects. This spike is also called an event. Each event consists of a timestamp with microsecond resolution, an event address represented by x and y pixel location, and the brightness change direction or polarity (0 means brightness decrease and 1 means increase). The DVS is data-driven rather than regular-sample-driven. It outputs a variable data-rate stream of timestamped pixel brightness change events.

Comparing with a conventional camera, the DVS has worse spatial resolution but better temporal resolution. The temporal resolution of DVS could go to $1us$, which is much faster than most super high-speed frame-based cameras. Thus, DVS asynchronously output an event stream while the frame-based vision sensors synchronously output the frame stream. It also has a higher dynamic range ($120db$ vs $60db$) and relatively low power consumption (mW at the die level).

Event cameras are available from these five companies: iniVation, Prophesee, Samsung, CelePixel, and Insightness. The comparison among these cameras refers to [11]. Here we give a brief history review on iniVation's cameras. The first generation of event cameras produced by them is called DVS128 [5], and as the name indicated, this camera has a resolution of 128x128. DVS128 was released in 2008, and it has a very large pixel size of 40um. Small resolution restricts many applications. However, as the first event camera presented in the research community, many seminal publications are still based on DVS128. Later in around 2014, the second genera-

tion of DVS called DAVIS240 [8, 12] was released. Compared with DVS128, DAVIS240 has a smaller pixel size (18.5um). Since then, DVS is upgraded to Dynamic and Active pixel Vision Sensor (**DAVIS**). A DAVIS [8, 9, 12] is a novel vision sensor that outputs asynchronous brightness change events concurrently with conventional frames. It combines DVS [5–10] and APS on one pixel. Therefore, DAVIS can also concurrently output intensity samples [7–10, 13, 14]. Besides that, the DAVIS added an Inertial Measurement Unit (IMU) chip centered on the same position as the sensor chip but the other side of the board. In summary, DAVIS provides three outputs: event stream, APS frames and IMU output. DAVIS346 is another event camera of the second generation. It adopted the same pixel size as DAVIS240 but has a higher resolution. The third generation is based on DAVIS346, but it provides additional color information with some color filters on the pixel.

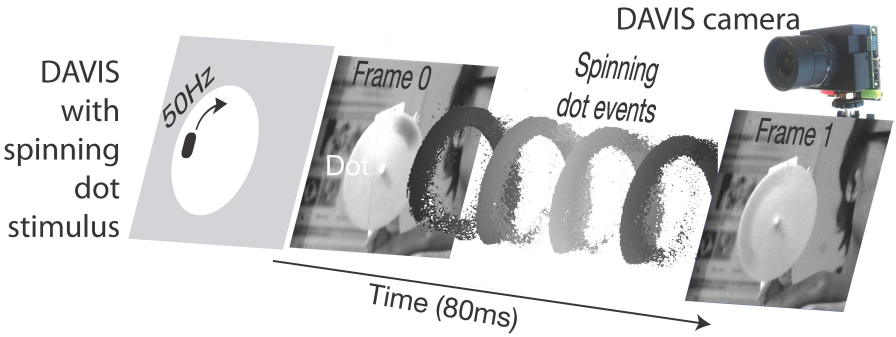


FIGURE 1.2: DAVIS's principle. It shows how the event stream and APS are generated by a DAVIS camera.

Figure 1.2 illustrates how DAVIS event camera works. The DAVIS camera is set in front of a spinning dot. This black dot is rotating on a white background with a frequency of 50Hz. Frame 0 and Frame 1 are generated by the APS part of the camera, and its interval is 80ms. The spiral dots between frame 0 and frame 1 are the events generated by DVS. It is evident that APS output is discrete, and some information is missed during this 80ms interval. However, thanks to the high temporal resolution provided by the DVS output, the event stream looks very continuous and can provide the trajectory of how this dot is moving during the "blind" period for APS frames.

In summary, the event camera, DVS or DAVIS provides high temporal resolution, sparse, and low latency data. It can work in a higher dynamic

environment and consume several mWs. It is recently widely used in all kinds of applications, from robotics, drones, even in satellites for star tracking [15, 16]. For low-level computer vision problems, it includes corner detection [17–20], segmentation [21], simulator [1, 22, 23] and dataset [24–27] The DVS has been proposed for high-level vision algorithms such as angular velocity estimation [28], tracking [29, 30], obstacle avoidance [31], landing [32–34], localization [35], navigation [36–38], VOD [39–42], and SLAM [43, 44]. For an overall review, we recommend readers to refer to [11]. More resources and code for event-based algorithms or applications are available on this GitHub repositorty[2]. However, more effort is required to exploit the advantage of DVS.

We reviewed the history of human perception of the world with video cameras. Muybridge started the first “motion picture" called zoopraxiscope. Since then, the cameras become more and more advanced while the prices are cheaper and cheaper. Cameras also walk out from the professional photography community and enter a variety of other areas. This results in the emergence of several other types of cameras. Event camera belongs to one of them and becomes increasingly popular in high-speed applications.

1.2 INTRODUCTION OF OPTICAL FLOW

OF is all around the world and exists in our life everywhere. When we walk in the street, we feel humans, buildings “moving" behind us. When we sit on a train, we feel the scene outside the window “moving" fast away from us. When we turn our heads, we feel the world rotating around us. All these movements we see caused by the environment and our motion in it are the OF.

Neuroscientists initially proposed the concept of OF. It was first studied in the context of neuroscience to understand motion perception in insects and mammals. James J. Gibson first proposed OF concept in 1950s. He describes the OF as the apparent flow of the movement of objects in the visual field relative to the observer and OF could be determined by using the pattern of light on the retina. The scenario of the moving object makes it as if the light flows in front of the eyes. Therefore, people vividly called it optical flow.

OF plays a vital role in insects. The primary sensor system of most insects uses some form of visible light to perceive the environment around

2 https://github.com/uzh-rpg/event-based_vision_resources

them. Visual navigation in flying insects is primarily based on the optical flow [45]. In essence, OF provides information on the ratio of velocity to distance, such that the actual metric distance to the environment is not directly available. Instead, flying insects navigate based on certain visual observables extracted from the optical flow field related to ego-motion. [46] shows that dragonflies could obtain the wide-field OF from their compound eyes. The biologists found that honeybees rely on OF for grazing landing [45, 47, 48], travel distance estimation [49], obstacle avoidance, and flight speed regulation [50].

In humans, OF combined with the vestibular system enables us to adapt the body's movements to the environment and thus to help us not to get lost and move safely in the world. The vestibular system in vertebrates is a sensory system that provides the leading contribution to the sense of balance and spatial orientation coordinated to movement with balance. The brain receives the OF and by interpreting the OF information, it could help humans/animals answer the following questions such as: Do I move or something in the environment? Which direction am I moving in? Am I moving straight ahead or am I turning? How far away are the objects from me? When will I encounter one of the objects? This information helps the human or animals localize themselves, measure distance, avoid collisions, *etc*.

Until now, researchers have found that the brain contains motion-sensitive neurons that react specifically to OF stimuli, translating the OF into neuronal spikes. However, despite OF is quite common for us, and we almost used it all the time, the mystery behind how the brain process OF information and thus answers the listed questions as we mentioned before is still under-investigated by researchers.

Computer scientists introduced the concept of OF to the computer vision area and used it to describe the relative motion between two consecutive images caused by the camera only, or only the observed moving objects or both of them. OF contains the changes between two images and thus contains the motion information of the natural objects projected on the image plane. Calculating OF makes it possible to recover the 3D motion in the real world of the observing object possible [51, 52]. For image and video processing, it is used in action recognition [53, 54] and video compression [55] and segmentation [56, 57]. In robotics area, it is also a very fundamental topic and it could be used for tracking [30], navigation [58–60], obstacle detection [61–63] and soft landing for flying drones [64–66].

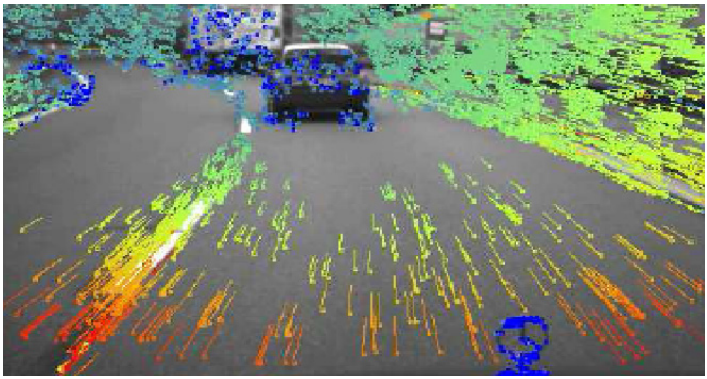


FIGURE 1.3: An example of the OF result in a road driving scene. Arrows represent the direction and the length means the magnitude. From website[3].

Figure 1.3 shows the OF result on a road driving scene. OF reflect the displacement between two adjacent frames. If the depth could be inferred, it is possible to convert this 2D OF to a 3D scene motion, and thus the velocity could be estimated. If the car is autonomous, the estimated velocity could help the car localize itself and drive more safely.

We also need to measure how good that vector is, and possibly such things as how different the next frame is. That is, the absolute difference between a pixel in frame 1 shifted by its motion vector and the corresponding pixel in frame 2. There can also be a confidence level or a penalty for how different a pixel is from its neighbouring vectors.

1.2.1 *Optical flow estimation methods for frame-based cameras*

The author of Middlebury OF Benchmark dataset Prof. Michael J. Black, wrote in his Ph.D. thesis: "Optical flow estimation is a chicken-and-egg problem: if you know how to segment the scene into differently moving objects, then computing their motion is relatively easy; if you know how to compute motion accurately you can segment the scene into differently moving objects. The problem is that these two things have to be done simultaneously. And it is hard." Dr. Black's thesis was finished in 1992. Almost

3 https://medium.com/building-autonomous-flight-software/math-behind-optical-flow-1c38a25b1fe8

30 years has passed, OF is a mature but not completely solved problem. This section gives a review of the frame-based OF estimation algorithms.

The OF estimation is based on two basic assumptions: i) brightness constancy assumption and ii) small movement assumption. To formulate OF in a mathematical way, that is to say, the goal of OF is for each pixel we need to calculate a motion vector which is a subpixel of x and y movement, say Δx and Δy. Based on assumption i, we can write the equation as follows:

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t) \tag{1.1}$$

And then applying the Taylor expansion for the right part of the equation, we get:

$$\begin{aligned} I(x + \Delta x, y + \Delta y, t + \Delta t) = I(x, y, t) + \frac{\partial I}{\partial x}\Delta x + \frac{\partial I}{\partial y}\Delta y + \frac{\partial I}{\partial t}\Delta t \\ + higher - order\ terms \end{aligned} \tag{1.2}$$

Based on the assumption ii, we can remove the higher-order terms and replace them back to equation 1.1, We get:

$$\frac{\partial I}{\partial x}\Delta x + \frac{\partial I}{\partial y}\Delta y + \frac{\partial I}{\partial t}\Delta t = 0 \tag{1.3}$$

Divide it by Δt:

$$\frac{\partial I}{\partial x}\frac{\Delta x}{\Delta t} + \frac{\partial I}{\partial y}\frac{\Delta y}{\Delta t} + \frac{\partial I}{\partial t}\frac{\Delta t}{\Delta t} = 0 \tag{1.4}$$

Use V_x and V_y to represent the flow speed in x and y directions:

$$\frac{\partial I}{\partial x}V_x + \frac{\partial I}{\partial y}V_y + \frac{\partial I}{\partial t} = 0 \tag{1.5}$$

Denote the brightness gradient of the image as I_x and I_y:

$$I_x V_x + I_y V_y = -I_t \tag{1.6}$$

There are two unknown variables V_x and V_y in equation 1.6 while only one constraint. To solve the equation, at least one more constraint is required. The two significant and pioneering works done by Horn and Shunk [67] and Lucas Kanade [68] in 1981 solved the problem with two different extra constraints. Because both of them are trying to solve the problem based on the gradients of the images, they are categorized into gradient-based methods.

Horn-Schunck [67] adds a global smooth constrain. It tries to make the flow field across the image smooth, so it adds an energy item to penalize the discontinuous OF result. The goal of Horn-Schunck is thus to minimize the following equation:

$$E = \iint \left[(I_x u + I_y v + I_t)^2 + \alpha^2 (\|\nabla u\|^2 + \|\nabla v\|^2) \right] dxdy \tag{1.7}$$

The minimum could be obtained by solving the associated multi-dimensional Euler–Lagrange equations. The positive side of Horn-Schunck is that it could provide dense optical flow output, and the negative side is that it is more sensitive to noise than the local methods. A range of modifications/extensions of Horn–Schunck, using other data terms and other smoothness terms, are proposed. For example, Black and Anandan [69] proposed segment smooth constraints, gradient constancy by Brox and Malik [70], color model by Mileva [71] and Zimmer [72].

Lucas Kanade added another constraint that the optical flow value in a small area stays the same. The motion vector between two images is estimated by dividing an image into patches of identical size, and it assumes that all pixels of the same patch have the same displacement. The patch size depends on the application. The equations for this method could be shown as follows:

$$\begin{aligned} I_x(q_1)V_x + I_y(q_1)V_y &= -I_t(q_1) \\ I_x(q_2)V_x + I_y(q_2)V_y &= -I_t(q_2) \\ &\vdots \\ I_x(q_{n-1})V_x + I_y(q_{n-1})V_y &= -I_t(q_{n-1}) \\ I_x(q_n)V_x + I_y(q_n)V_y &= -I_t(q_n) \end{aligned} \tag{1.8}$$

Where $I_x(q_1)$, $I_y(q_1)$ and $I_t(q_1)$ are the partial derivatives of the image intensity with respect to position x, y and time t, evaluated at the point q_1 and at the current time. Points q_1, q_2 $\ldots q_n$ are within the same patch region. Equation 1.8 could be solved by the least-squares principle.

After about 30 years of development, there is a massive improvement of the techniques used for different aspects of optical flow estimation. All these methods could be categorized into four types: block matching [55], phase-correlation, gradient-based and learning-based.

Block matching is a method that originated from MPEG compression. The core idea of this method is to find the best similar region to the reference region of frame 0 in frame 1. The motion vector is searched in

a defined search window for each region to minimize the criterion chosen [73]. The criterion used to measure the similarity differs for different applications, but the most frequently used in the research is SAD. The most straightforward search strategy for block matching is the exhaustive method which compares all possible blocks with the reference block. Some other non-exhaustive search strategies [74–77] are also explored. However, no matter which kind of strategy is used, complicated math operations such as exp or derivatives are not required, making it very hardware-friendly. The disadvantage of this method is that the accuracy of this method is not so good, and sometimes you get lots of spurious matches.

Phase correlation is an approach to estimate the relative translation offset between two similar images (digital image correlation) or other data sets. OF based on phase correlation firstly converts the image from the time domain to the frequency domain. Then it calculates the cross power between the frequency spectrum of these two images. OF is estimated when it maximizes the normalized cross-correlation, which is obtained by applying the inverse Fourier transform to the cross power result. Compared with spatial-domain algorithms, the phase correlation method is resilient to noise, occlusions, and other defects typical of medical or satellite images. Therefore, it tends to give more accurate results. Although it looks pretty complicated, as it requires lots of FFT's and multipliers, it is rather hardware-friendly as it is a closed-form solution that does not require iteration.

From the aspect of accuracy, the gradient-based method might be the most accurate method among non-learning-based methods. The idea is to take two blocks of data from adjacent frames and to use the gradient information to progress iteratively to a solution where the pixels are aligned. This gradient information includes partial derivatives of the image signal and/or higher-order partial derivatives. Two methods Horn Shunck [67] and Lucas Kanade [68] we introduced before, belong to this category. The gradient-based method is usually combined with a hierarchical architecture or called CTF. The reason for that is when the displacement of two images is too big, the gradient-based method does not perform very well. Although it gives a more reliable result, multi-scale and requirement of derivative calculation make it highly computationally expensive. Other gradient-based work includes [46, 78, 79].

Recent years witness the rapid development of deep learning. There are many latest OF is based on learning architecture. Weinzaepfel *et al*. [80] developed a method called DeepFlow. The name seems like a deep learning-

based approach, and it is not. However, it has similar ideas as Deep Neural Network (DNN). It uses convolutions and pooling similar to a CNN and creates a pyramid of images. The correspondences were first found in the highest level and then repeated until all levels' correspondences were tracked. Revaud *et al*. [81] developed EpicFlow, which produces higher accuracy on optical flow evaluation from the Sintel dataset. The algorithm uses a novel interpolation method robust to large pixel displacements, boundaries of motion, and occlusions within the image. However, this method runs very slowly. Its run time exceeds 16.4 seconds for a MPI-Sintel image pair (1024 $\times$ 436 pixels) on one CPU-core at 3.6Ghz. Fischer *et al*. [82] created FlowNet, a fully convolutional deep neural network trained to produce optical flow images from a pair of input images. It is the first-ever deep learning scheme based on convolutional neural networks. However, its efficiency remains below state-of-the-art traditional methods. FlowNet2 [83] built a more robust structure by stacking together multiple FlowNet modules. It achieves better performance than the original Flownet, but it requires heavy memory (around 38 million parameters); hence is not well suited for mobile and other embedded devices. A light version of FlowNet called LiteFlowNet [84, 85] is proposed in 2018, and its improvement version in 2020 by Hui. Ranjan *et al*. proposed SpyNet [86], which combines the CTF approach with DNN and it has 96% less weights than FlowNet. PWC-net [87] uses pyramid, warping and cost volume. It adopts a similar idea as SpyNet but with some differences: i) SpyNet warps frames with coarse estimates of the flow, while PWC-net warps feature maps. ii) PWC-net data augmentations do not include Gaussian noise [88]. Other CTF deep-learning-based optical includes [84, 89]. To date, deep learning, especially the CNN, has led to breakthroughs in frame-based optical flow area. However, this advance comes with major computational demands due to the use of cost-volumes and pyramidal representations. Cost-volumes is a concept initially from stereo matching and later used in OF to represent the matching cost between two feature volumes.

For the dataset and benchmark part, the community has accumulated a lot of resources [82, 90–97]. Among them, the most popular evaluation benchmarks are Middlebury, MPI-sintel and KITTI. The Middlebury benchmark [90] is composed of sequences partly made of smooth deformations, but also involving motion discontinuities and motion details. MPI-sintel [91] is drawn from a short computer-rendered movie. It counts around 1500 frames with OF groundtruth. It is a challenging benchmark including fast motion, occlusions, and nonrigid objects. KITTI [92, 93] has two

versions, and the latest one was published in 2015. It is tailored for autonomous driving. Compared to KITTI2012, it contains more dynamic scenes, large motion, illumination changes and occlusions, and HD1K dataset.

To summarise the frame-based OF algorithms, the problem of optical flow estimation could be considered as almost completely solved [98] in the particular case of small displacements. And the accuracy of OF methods is actually pretty good except at motion boundaries. The remaining challenges can be listed as (i) fast motion, (ii) illumination changes, (iii) occlusion, and (iv) untextured regions. In these challenges, the optical flow estimation problems become ill-posed and hard to treat analytically.

1.3 OPTICAL FLOW APPLICATIONS

OF is widely used in a variety of applications, and some of them occur almost every day. This section will show three types of optical flow applications as examples: film, optical mouse, and robotics.

1.3.1 *Use of optical flow in the film production*

OF plays an important role in creating visual effects in the film industry. In the world of visual effects, OF is used as a tool for re-timing shots, tracking, 3D reconstruction, and motion blur.

OF technology was first used in feature films starting from 1995. Kodak's American developed a new image technology called Cinespeed which could interpolate between frames based on OF. They contacted Kim Libreri, known for his work on the *Matrix* series and *Super 8*, who was Chief Technology Officer of a visual effects company called Cinesite London. Kodak introduced this new technology to Libreri and thought it might be helpful for them. The technology was quickly named Cinespeed and was the first major commercial OF re-timer as part of the Kodak Cineon System. Cinespeed was later used in the film *Mission: Impossible* and made this film the first film shot using OF re-timer technology. One scene in which the camera circled Tom Cruise when he kissed Emmanuelle Beart on a rough turntable used this technology. It was successfully smoothed and re-timed using a combination of traditional 2D algorithms and Cinespeed. "We were basically presented with a working prototype of a system for re-timing" recalls Libreri, "while the shot was cut from the final film, it worked well." Although this shot was cut from the film in the final release, it worked very well and inspired people to consider OF for the film shooting.

FIGURE 1.4: Bullet time effect in *Matrix*. From website[4].

The real breakthrough for OF used in the film came in 1998. This film is called *what dreams may come* which won the Academy Award for Best Visual Effects in 1998. In this film, the male protagonist, acted by Robin Williams, entered a "virtual heaven world" after his death. This heaven world is based on an oil painting which his beloved wife draws. Therefore, many scenes of the film are virtually painted drawings. In the film, he is live-action, but it behaves like it is made of paint when he touches the foliage. This visual effect is achieved with OF tracking techniques. Synthetic objects could be driven by the captured motion and placed into the live environment with the OF. The brush stroke tracking systems help process the images for a painterly effect and animate the brush strokes to follow the underlying motion in the scene. The final effect works quite well, and it makes the filmmakers and artists think of it not only as a re-timing tool and can be used for some other interesting applications such as tracking brush strokes and 3d plants. Tracking the paint stroke is not the only usage of OF. It is also used for adding motion blur to the painted strokes, 3d elements, and warped 3d models. In 1998, a REVisionFX press commented that *what dreams may come* marked an important "first" in visual effects, "successfully implemented the first production pipeline to use computer vision technology so pervasively. The 'Painted World' section is the first long moving picture sequence that relies on image-based animation and

4 https://www.telegraph.co.uk/films/2019/07/24/inside-keanu-reevess-bullet-time-scene-matrix-changed-cinema/

non-photorealistic rendering, two growing areas of development in computer graphics." This visual effect work is done within Mass Illusion.

FIGURE 1.5: The history scene that the actor meets his younger counterpart. The man sitting on the left is the real actor and the right side man is the synthesized person. Figure courtesy of [99].

The other example that is important to show OF used in the feature film industry is the bullet-time effect in the film *Matrix*. Many people who watched *Matrix* might be impressed by its shocking visual effect. There was a very famous scene in which Keanu Reeves leaned himself to avoid the super-fast bullet. The whole process is shown in a super-slow-motion way to make the audience see it clearly and more immersive to the film scene. This visual effect was called the bullet-time effect and later became a comprehensive popular technology in the film. Fig 1.4 shows how the bullet time effect looks like in *Matrix*. The audience can even see the shock wave generated by the super-fast bullet in the film. Multiple cameras in the film were arranged in orbit, which a pre-defined rig path was forming a complex curve through space. The path was designed based on computer-generated visualization. The multiple cameras on the rig were triggered in a very short interval, and because the cameras are arranged in a shape similar to a spiraling up circle, the viewpoint was thus changed. Additionally, the individual frames were sent to the computer for further interpolation where the technology behind was the OF algorithm. This approach

improves the fluidity of the movement and thus generates the super-slow-motion scene. The bullet time effect in the *Matrix* promotes the super-slow-motion shooting for the follow-up films. Following films, Bullet time was also featured as crucial gameplay mechanics in various video games such as Max Payne and Cyclone Studios' Requiem: Avenging Angel.

FIGURE 1.6: The motion blur visual effect. The motion blur behind the flash man is used to emphasize his super fast speed. This is a poster for the *Flash* movie series. From website [5].

The Disney human face project is another interesting use of OF. This project constructed a virtual person who was the young version of the real actor, and these two "guys" were sitting side by side in the film. The visual effects team for this film is from Walt Disney Feature Animation. To achieve this goal, they first shot the actor's facial performance against the green screen. Then, they ran the OF to track each pixel's motion over time in each view. A cyber scan model of a neutral expression of the actor was combined with the OF result and the photogrammetric reconstruction of the camera positions. After the 3D model for the target actor was established, it projected a vertex of the model back into the 2D image of the "virtual" young face and then tracked the motion using OF again. The OF algorithm they used was based on Black's work [69]. It solved the OF in an iterative way and tracked the eye in a manner analogous to Lucas-Kanade affine tracking. The algorithm finally worked quite well. The track result was then rendered on the young face model. In 2002, they presented a

5 https://hdqwalls.com/wallpaper/3840x2400/the-flash-run-art

demo of their work [99] at SIGGRAPH and attracted a lot of attention. Fig 1.5 shows the final visual effect in the film.

Motion blur visual effect is another common application of OF for films. Fig 1.6 shows an example of how the motion blur visual effect looks like. The scene in this figure is from a London street. This visual effect is also achieved by OF. The creator frequently uses it to emphasize some objects. It also helps heighten the sense of reality.

After about 20 years of development, OF is now in common use in a variety of commercial productions and almost becomes a standard technology used by many big visual effects companies such as ReVisonFX, Foundry, ILM, *etc*.

1.3.2 *Use of optical flow in computer mice*

According to the working principle, computer mice can be divided into two categories. The first category is the mechanical mouse. This is also the primary type of the earliest mice. There is a ball installed on the bottom of the mouse. By measuring the distance the ball rolls, the mouse could convert it to the value the arrow on the screen should move. The accuracy of the mouse entirely depends on the precision of the ball. Therefore, the mechanical mouse requires cleaning the ball frequently and thus makes it not very convenient.

The second category of mice, "optical mouse," was thus developed to solve problems of the mechanical mouse. The optical mouse, more accurately, should be called optical-electronic mouse since the core part of the mouse is a circuit implementing the OF estimation algorithm. OF, especially the global translation flow, is a fundamental measurement for an optical mouse. Fig. 1.7 shows the principle of the optical mouse. As shown in this figure, the optical mouse mainly consists of three parts: LED, lens, and OF sensor. The environment under the mouse is very dark, and it is challenging to get the images there. The LED is thus installed to lighten it and combined with a lens to make the mouse could see it clearly with the help of a small lens. The OF sensor is often made up of an Image Acquisition Sensor (**IAS**) and a Digital Signal Processing unit (**DSP**). Usually, there are two chips on the Printed Circuit Board (**PCB**). The IAS outputs the images of the observing surface or texture. The resolution of the image sensor is often not very high, such as 32x32. DSP is usually a microprocessor such as a low-cost ARM with OF algorithm that has been flashed inside. It extracts the motion information between these two matrices based on the pixel lo-

cation and intensity information. After the OF is obtained from the sensor, the mouse could measure the actual translation distance and could help it localize and update the coordinates of the mouse arrow on the PC screen. The position information is sent to the PC via PS2/USB (wired mouse) or 2.4Gz radio/Bluetooth (wireless mouse).

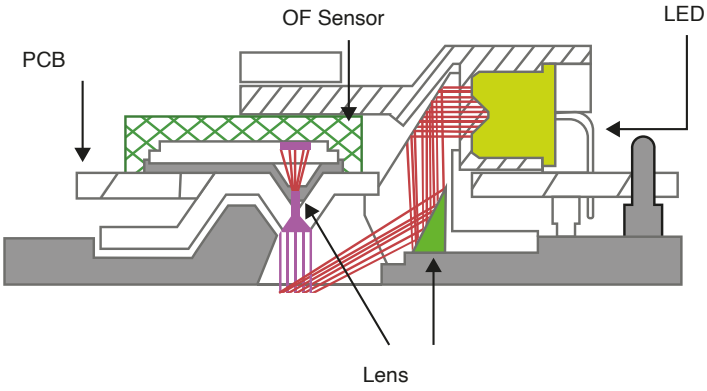


FIGURE 1.7: The working principle of the optical mouse. It mainly consists of three parts: LED, lens and OF sensor. Adapted from [6].

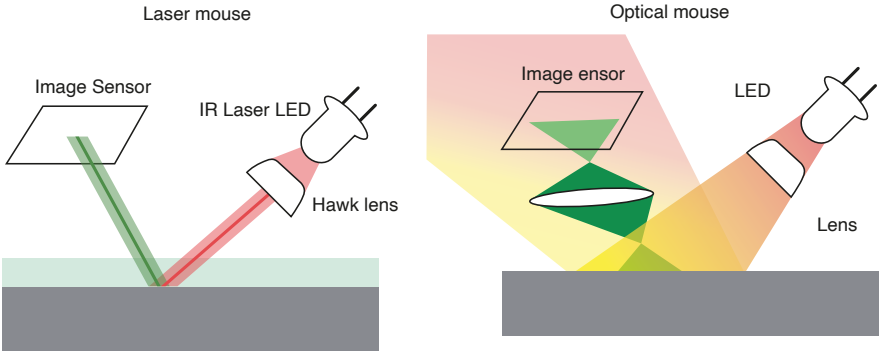


FIGURE 1.8: Difference between IR mouse and optical mouse. Adapted from [6].

During the early days, regular LED, especially red LED, is the primary light source in the optical mouse. However, this light beam of this LED source is very dispersed and makes the light intensity arriving at the sensor weak. The diffuse reflection even makes it worse. To get clear images

6 https://kknews.cc/zh-sg/news/23mk5e.html

from IAS, the LED's intensity should be strong enough. On some specific surfaces, such as glasses, it is almost entirely dark and cannot work at all.

To address this problem, the laser replaces the regular LED as the light. Fig. 1.8 compares the difference between the normal optical mouse and laser mouse. As the laser source is coherent light only consists of one single wavelength, it could maintain the same intensity and wave shape even after a long-range transmission. As the laser light is concentrated, it also helps avoid diffuse reflection.

However, no matter which light source is used, the core computation parts of them are the same. The OF algorithm is still the essential software for a mouse.

The metrics to evaluate the performance of a mouse are resolution and refresh rate. The resolution is also called Count Per Inch (CPI). It describes how much feedback the mouse could receive from the DSP by mouse. For example, if a mouse has a CPI 400, it means that when the mouse moves an inch in the physical space, it will receive 400 measurement times feedback. In other words, the minimum motion displacement it could sense is 1/400 inch. This metric is used to represent its spatial resolution. The CPI is mainly determined by the minimum OF vector displacement the DSP and image sensor could obtain. Higher CPI is more suitable for higher resolution screens. The other metric is called refresh rate. This one is easy to understand. It is very similar to the FPS in computer vision. The higher the refresh rate, the higher accuracy the mouse could obtain but impose faster and low power consumption OF algorithms.

1.3.3 *Use of optical flow in microrobotics*

Rapid advances in microelectronics catalyzed the development of tiny flying robots, formally referred to as Micro Aerial Vehicless (MAV) [100]. OF estimation has always been a low-level but fundamental topic in machine vision; it is widely used in take-off and landing, 3D reconstruction, and navigation in the robotics community. [101] presents complex maneuvers to be able to perform the autonomous take-off and landing based on an optical flow sensor. Some event-based OF used for landing is demonstrated in [32, 34].

Many types of vision sensor hardware have been used to collect videos of optical flow in robotics. The sensor needs to be small and light due to the constraints of small or microrobotics platforms. More importantly, the hardware limitations of different vision sensors can significantly affect the

consequent optical flow computation. For instance, the pixel count directly affects the computational power required for on-board optical flow computation, while the field of view affects the sensing range of the robot. Thus, the optical mouse sensor is a very straightforward option. A mouse sensor-based optical flow module for quadrotor control is shown in [102]. [59, 103, 104] use an optical mouse sensor for VOD. But using a mouse to estimate optical flow for MAV has many drawbacks. It is unsuitable for indoor applications since standard mouse sensors require stronger lighting than in normal indoor conditions. These mouse sensors work well with in-door well-conditioned environments, e.g., surfaces with sufficient texture for motion detection [103]. [105] proposed a customized hardware optical flow-based to solve the illumination limitation problems of mouse optical sensors. This one could work well on a wide range of illumination environments but cannot work on the constant distance plane. Optical mice generally output only the 2d global translational flow vector, which is not sufficiently informative for motion parallax or obstacle detection or VOD.

[36, 106] shows a flight control system using a CMOS image sensor for OF computation. However, the data is not processed on board; it is sent to a computer on the ground using a wireless link. The processed OF result value is sent back to the MAV.

They use pix4flow [105] as the algorithm module. *pix4flow* [105] is an embedded OF module for real-time computation. The size of the module is 45.5 mm x 35mm and consumes about 0.5W. It uses the ARM Cortex M4F as the processing unit and connects to the sensor via a parallel interface. The image sensor is a 752x480 MT9V034. It processes about 60 frames for the full resolution. In [105], they use a gyro to estimate the angular velocity, and then it is used to compensate the OF. The camera translation part of the OF is extracted, and a metric value which is the depth is used to scale it back to the actual velocity. The depth is assumed approximately constant since it is tested in a hovering condition. The distance from the sensor to the ground is almost unchanged.

1.4 OPTICAL FLOW SENSORS

Optical flow sensors are used extensively in the computer optical mouse as the primary sensing component for measuring the mouse's motion across a surface. We already show this part in in Sec. 1.3.2. Optical flow sensors are also being used in robotics applications, primarily where there is a need to measure visual motion or relative motion between the robot and

Sensor type	Image pixel	Size	Weight	Update rate	FOV	Applications
ADNS-2610 [107]	18 x 18 pix.	25 x 30 mm 35 x 30 mm 50 x 30 mm	15 g 23 g 23 g	1500Hz 20Hz	6.5deg 2.5deg 1.2deg	Obstacle avoidance
CentEye TinyTam	16 x 16 pix	7 x 7 mm	125 mg	20Hz	N/A	MAV Hovering
OV7740 + FPGA [108]	320 x 240 pix	6.5 x 6.5 x 4 mm		120Hz	50 deg.	On board OF calculation
PX4FLOW [105]	188 x 120 pix	45.5 x 35 mm		250/500Hz	21 deg.	On board OF calculation
GoPro Hero [109]	1920 x 1080 pix	42 x 60 x 30 mm	94g	29.97Hz	127 deg.	Compare w. navigation sensors

TABLE 1.2: Comparison of different hardware optical flow sensor. Adapted from [59].

other objects in the robot's vicinity, such as Zuffrey *et al*. [110], who used it for for stability and obstacle avoidance. About this part we have described in Sec.1.3.3.

An optical flow sensor is a module capable of capturing images and measuring OF information. The module could be a small PCB consisting of several chips, as we often see in the mouse. It can also be a more compact version that integrates the image sensor circuit and the processing circuit on the same die, forming an Application Specific Integrated Circuit (**ASIC**) [111, 112]. The processing circuitry may be implemented using analog or mixed-signal circuits to enable fast optical flow computation with minimal power consumption. One area of contemporary research is the use of neuromorphic engineering techniques to implement circuits that respond to optical flow, which may be appropriate for use in an optical flow sensor [113]. Such circuits may draw inspiration from biological neural circuitry that similarly responds to optical flow.

A comparison of the characteristics of various optical flow sensors is shown in Table 1.2.

1.5 THESIS CONTRIBUTIONS

As shown in Sec. 1.1.3, the DAVIS event camera provides sparse, quick, high dynamic range events signaling brightness changes as well as global-shutter APS image frames that can be triggered and captured on demand.

It also integrates an IMU that provides vestibular rotation and acceleration sensing. These characteristics make it potentially useful for mobile robotics, but it almost provides practically no on-board computing ability. Although the USB device controller includes an MCU (Cypress FX2 for DAVIS240 and FX3 for DAVIS346), it has no access to the data, which flows directly through the USB Integrated Circuit (IC) via the USB chip endpoint First In First Out memories (FIFO). Moreover, the Complex Programmable Logic Device (CPLD) logic chip on the DAVIS PCB that interfaces between the USB chip and the DAVIS has limited resources and is almost filled by the its logic circuits for Address Event Protocol (AER) and APS data handling. Therefore, the first achievement of my Ph.D. project is that I designed a new DAVIS camera that features the DAVSI346 sensor and a SOC that includes a powerful FPGA and a dual-core 800MHz ARM processor (Xilinx Zynq 7z100). The new camera is called `DAVIS346Zynq`.

The computer vision community has explored frame-based OF for a long time. However, due to the disadvantage of the frame-based camera compared with event cameras, such as low dynamic range and low temporal resolution and too much redundant data, in some complicated environments such as fast motion or high dynamic range light conditions, the result is not good enough. Almost all developments are based on camera frames. These frame-based OF algorithms are thus not able to be applied to event cameras directly.

Compared with frame-based camera OF, event-based OF could obtain the samples at a high frequency of up to several kHz effective rate. This makes the "brightness constancy constraint" underlying most OF computations more feasible. Events are generated in continuous time and could resolve the large displacement and motion blur problems that occur in frame-based OF. Therefore, event-based OF receives increasing attention from many research communities, not only from the computer vision side but robotics. In this thesis, we proposed several novel algorithms dedicated to event-based cameras and took the limitations of real robotics into account. One is for OF estimation and the other is for corner detection. Combining the corner detector with the optical flow estimation achieved a good accuracy while maintaining a low power consumption, low memory occupation, and enabling real-time operation even at high event rates of over 10 MHz. It has been implemented and verified on `DAVIS346Zynq`.

In summary, the main contributions of this thesis are:

- A novel event-based OF method is called ABMOF (Chapter 3). The incoming event actively drives the estimation. Events are accumulated

into rotating event slices. The duration of the event slice is controlled by a feed-forward method named *AreaEventNumber* and a feedback mechanism adapts the event exposure count value to scenes with various levels of spatial sparsity.

- A hardware corner detection method called SFAST (Sec. 6.3.2) which is more accurate, more compatible with ABMOF and more efficient than the prior hardware EFAST corner detector (Chapter 4).
- A powerful event camera `DAVIS346Zynq` (Chapter 5) that targets real-time applications. It features the most powerful SoC Zynq 7100 FPGA among the Xilinx Zynq families. It has a DAVIS controller for AER protocol handling, a USB controller, and a VGA port. For memory, it supports 512MB DDR3 and 1GB NAND flash.
- An open-source hardware implementation of combined SFAST and full multiscale adaptive event slice ABMOF, which together improves the accuracy of optical flow estimation and increases the throughput (Chapter 6). It combines several techniques in scheduling and Hardware Description Language (**HDL**) architecture (*e.g.*, data partitioning, on-the-fly processing with minimal buffers, deep pipelining, parallelization at bit-/data-/instruction-/task-level) for efficient circuit design.

1.6 THESIS STRUCTURE

The remainder of this thesis is organized as follows. Chapter 2 describes the basic event-based Block Matching Optical Flow (**BMOF**) algorithm and its FPGA implementation. Chapter 3 shows an improvement version of BMOF called ABMOF. The event-based corner detectors EFAST and its FPGA implementation are introduced in Chpater 4. Chapter 5 is concerned with the `DAVIS346Zynq` camera. Chapter 6 presents the EDFLOW which implements the corner detector and ABMOF on `DAVIS346Zynq`. Chapter 7 concludes this body of the thesis and provides an outlook for the possible future directions. Appendix A provides the description of the MiniZed Software-Defined System On Chip (**SDSoC**) platform, High Level Synthesis (**HLS**) optimization strategies, troubleshooting stories, UART configuration of `DAVIS346Zynq` and URL links to resources.

The chapter that follows now describes my earliest and simplest developments of DVS OF where we first explored the use of block matching for measuring OF.

2

A FIRST EVENT-BASED BLOCK-MATCHING OPTICAL FLOW ALGORITHM AND ITS FPGA IMPLEMENTATION¶

> *"Analytically, this total transformation of the array appears to mean that the elements of this texture are displaced, the elements being considered as spots. Introspectively, the field is everywhere alive with motion when the observer moves."*
>
> — James J. Gibson, 1966

This chapter describes my first DVS OF algorithm which uses a BMOF approach and the FPGA implementation of this algorithm.

2.1 INTRODUCTION

This section has two parts. The first part explains why event-based OF is important and what are the challenges of measuring OF. The second part reviews relevant prior event-based OF work.

2.1.1 *Why event-based optical flow*

In Chapter 1, we introduced the definition of OF in neuroscience and computer vision, and reviewed some algorithms for frame-based cameras (see Sec. 1.2.1). Much work and research has been done in OF for frame-based cameras. However, the fixed sample rate of frame-based cameras makes it difficult to get small-displacement image pairs when the scene is moving too fast. For example, the most popular method for frame-based OF is Lucas-Kanade (LK) proposed in 1981 [68]. It works well on the small displacement motion but badly on scene with large motion flow (see Fig. 3.10 in Chapter 3). LK also fails when the images are over or underexposed and when the images are too blurred to extract good features. The DVS's super high temporal resolution, low latency, and high dynamic range could provide solutions to these problems.

¶ A substantial content of this chapter is published in [114].

However, event-based OF is also challenging because of the unfamiliar way in which events encode visual information. In conventional cameras, the optical flow is obtained by analyzing two consecutive images. These provide spatial and temporal derivatives that are substituted in the brightness constancy assumption (Equation 1.3), which, together with smoothness assumptions, provide enough equations to solve for the flow at each image pixel. In contrast, events provide neither absolute brightness nor spatially continuous data. Each event does not carry enough visual information to determine the flow, so events need to be aggregated to produce an estimate, leading to the unusual question of where in the x-y-t-space of the image plane spanned by the events is flow computed [11]. Ideally, one would like to know the flow field over the whole space, which deems computationally expensive. In practice, optical flow is computed only at specific points: at the event locations or artificially chosen points. Finally, another challenge is to design a flow estimation algorithm that is compatible with what is known from neuroscience about early processing in the primate visual cortex, and that can be implemented efficiently in neuromorphic processors. Nevertheless, computing flow from events is attractive because they represent strong spatial gradient area, which are the parts of the scene where flow estimation is less ambiguous, and because their fine timing information allows measuring high-speed flow [115]. Meanwhile, DVS provides high dynamic range and activity-driven brightness change events, and it makes possible to explore some environment which is difficult for conventional cameras.

2.1.2 *Prior DVS optical flow*

This section reviews previous DVS OF algorithms of which there are only a handful. Event-based OF has been proposed to date. It could be classified into three categories. The first category adapts the conventional OF algorithms to event-based versions, such as event-based Horn Schunck and event-based Lucas Kanade. The second category is the variational method, such as motion compensation framework [116]. The third category is the learning-based method, such as *EV-flownet*.

[117] described an open-source algorithm (called Direction Selective (DS) in this thesis) for time-of-flight DVS OF based on oriented edges detected by spatio-temporal coincidence. It works only for sharp edges and suffers from aperture problems since it is edge-based.

[118] adapted the frame-based LK algorithm (called Event-based Lucas-Kanade (EBLK) in this thesis) for the DVS. It stores a fixed-queue length window of past events. For each new event, it computes the LK algorithm on a window of the fixed time interval of a block of pixels surrounding the current event pixel. Brightness of the pixel is replaced by the event count and the gradient calculation is thus based on the quantized event count. The gradient estimation precision is low due to quantization and a small 5x5 window size. The small window size was used to limit the computation time in order to keep up with a high rate of events.

[119] proposed a contour-based method. In their work, they compared the events-only method and events-frames-combined method. The difference is that by using the only the events, they need to reconstruct the contrast of the edge to localize the contour but it is not necessary for frames that have the absolute intensity. The optical flow estimation is then obtained from the contour width divided by the time interval.

[120] proposed a time-surface method (called Local Plane (LP) in this thesis) that combines the 2D events and timestamps into 3D space. Normal OF is obtained by robust iterative local plane fitting. It works well for sharp edges but fails with dense textures, thin lines, and natural scenes [115, 121] since all these produce complex structures that plane fitting does not model.

[122] proposed a more expensive phase-based method for high-frequency texture regions. They use normalized cross-correlation to measure the pixel's timestamps' similarity and localize the contour. Once the contour is found, they use a Gabor filter to extract the local phase. The OF constraint that assumes the constancy of the spatio-temporal contours using the phase is formulated and is used to solve the normal OF flow.

[115] implemented and compared the DS, EBLK, and LP methods. It concluded that the existing algorithms were both computationally expensive and do not work well with natural scenes and noisy sensor data. This paper also proposed an evaluation method and provided a simple benchmark dataset with ground truth. The ground truth OF is obtained by constraining the camera motion to pure rotation and uses the camera's IMU rate gyros to obtain the global translational and rotational OF.

[123] proposed a frame-based variational algorithm that simultaneously estimates the OF, gradient map, and intensity reconstruction from DVS. Although the simultaneous constraints results in a regularized output, the results are not quantified, and the method is very computation expensive compared to others.

Although the main goal of [29] is for event-based feature tracking, it also proposed a pipeline to compute OF on corner points. They added another two assumptions: One is that events generated by the same point lie on a curve, and OF within a small spatio-temporal window is constant. The OF problem is cast in an optimization framework, and the expectation maximization algorithm computes the solution. It can run in real-time with 15 features on a PC.

In recent years, motion compensation combined with optimization approaches have been widely used for event camera OF [28, 116, 124, 125] and have become the most popular approaches among all the first and second category OF methods. These methods stem from the simple observation that if the event cloud is viewed in 3D spacetime, there is a view angle where the events locally line up. This view angle represents a particular flow. The optimization procedure to locally align the event cloud can be carried out by a search procedure that maximizes the contrast of the resulting 2D image. Since it requires multiple local optimizations and lots of memory to hold a cloud of events, it needs an entire desktop Graphics Processing Unit (GPU) for tens of FPS.

Other event-based OF approaches were also developed. The relation between segmentation and OF is like the chicken and egg problem: better segmentation improves OF accuracy and more accurate OF improves segmentation. In [125], they analyzed the influence of different contrast maximum reward functions on the aperture problem to jointly estimate the pixel-level segmentation and OF. [126] estimated the OF and intensity image from a single blurred DAVIS APS image and event stream, similar to [123]: both use variational methods to estimate OF and intensity jointly. However, [126] exploited single-frame motion blur, so they only require one single APS image. Nevertheless, both of them are very slow, *i.e.*, less than 1 FPS on desktop CPU. For example, [126] was implemented in Matlab using C++ wrappers and it takes around 1.5s to process one image on a single i7 core running at 3.6GHz, which is very far from real-time processing.

Akolkar [127] proposed an optimal method based on [120]. It divides a curve into several segments and adjusts the scale to maximize the mean value of all segments' normal optical flow. Using iteration can help the algorithm find the best scale of the window size to mitigate the aperture problem. It obtains the actual flow rather than the normal flow. Low [128] is another work based on LP fitting. It proposes using Prim's algorithm to find the optimal event sets for plane fitting to improve the accuracy.

They simplify the original LP fitting [120] by imposing more constraints on the incoming event and make it a non-iterative. Guidelines for potential implementations on hardware are also reported in this work. However, it cannot solve the essential problem that the original LP method faced, which is that the result is still the normal flow and still suffers from the aperture problem.

Deep learning is also explored in event-based OF algorithms [129–132]. *EV-flownet* [129] reported the first CNN-based DAVIS OF architecture and published a valuable dataset. The network is trained by minimizing the photometric loss from the DAVIS APS frames. It achieves the best-published accuracy, but burns 50W to run at 25 Hz frame rate on a laptop gamer GPU. They further extended this network to jointly estimate the depth and ego motion together in [133]. These DNN learning-based methods are more accurate than the hand-crafted approaches, but they are much more computationally intensive. For example, [134] reported that their most capable CNN runs at 40 FPS on an NVIDIA 1080Ti GPU whose power consumption is about 200W.

[135] proposed another unsupervised brain-inspired learning rule. They presented an adaptive STDP rule to estimate the global OF from local OF estimations. They use Spiking Neural Network (SNN) as the network architecture. Even though the accuracy is not good as the CNN network architecture used by Zhu *et al*. [24, 133], it introduces methods of using SNNs for OF learning.

Most of the works mentioned above are based on PC computation [115, 118, 120, 122]. There is one work is based on embedded system. In 2015, Conradt *et al*. [136] proposed a real-time optical flow algorithm based on DVS which is implemented on an ARM 7 microcontroller. Although Conradt's work [136] achieved a real-time result, it was only characterized for camera rotation, not camera translation through space, and its use of the direct time of flight of events makes it unlikely to work well with densely textured scenes and to suffer from aperture problems for edges.

In this chapter, we introduce an event-based BMOF algorithm and its FPGA implementation. This chapter is organized as follows: Sec. 2.2 introduces the system architecture and algorithm. Sec. 2.3 shows experimental results, and Sec. 2.4 concludes this chapter.

2.2 BMOF ALGORITHM AND ITS FPGA IMPLEMENTATION

The output of DVS is a stream of brightness change events. Each event has a microsecond timestamp, a pixel address, and a binary polarity describing the sign of the brightness change. Each event signifies a change in brightness of about 15% since the last event from the pixel. In this work, events are accumulated into time slice frames as binary images ignoring the polarity, since our aim is for minimum logic and memory size. Here we will refer to these bitmap frames as event slices. It is equivalent to the concept event frame used in [14]. For clarity, we use *event slice* in the following text.

In video technology, OF is called Motion Estimation (ME) and is widely used in exploiting the temporal redundancy of video sequences for video compression standards, such as MPEG-4 and H.263 [137]. The pipeline for ME includes block matching. Block matching means that rectangular blocks of pixels are matched between frames to find the best match. Block matching is computationally expensive. That is why it is now widely implemented in dedicated logic circuits. To address this problem, an example of logic ME implementation based on block matching is presented in Shahrukh [137]. In this chapter, we proposes an event-based block matching algorithm to calculate OF on FPGA.

A block is a square centered around the incoming event's location. Matching is based on a distance metric. In this work, we implemented Hamming Distance (HD) as the distance metric. HD is the count of the number of differing bits. For bitmaps, HD is exactly the same as the better-known SAD. The software implementation is open source.

2.2.1 *System architecture*

The hardware evaluation system is divided into two parts, one for data sequencing and monitoring and the other for the algorithm implementation. For the first part, we use a monitor-sequencer board [138] designed by Raphael Berner in his masters project at Univ. of Seville . The sequencer converts the event-based benchmark dataset [115] into real-time hardware events sent to the FPGA for OF estimation. During OF calculation, the monitor collects the OF events and sends them over USB to jAER for rendering and analysis. In this way, we can compare the software and hardware processing of the OF algorithm. In this work, we only used prerecorded data to allow a systematic comparison between software and hardware implementations.

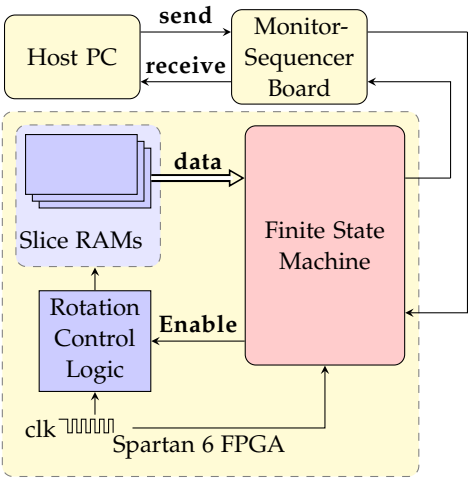


FIGURE 2.1: System Architecture

The OF architecture (Fig. 2.1) contains three main modules: the Finite State Machine (FSM), block RAMs and rotation control logic. The architecture of the FSM is shown in Fig 2.2. The FSM consists of three parts: data receiving module, OF calculation module, and data sending module. The data sending and data receiving module communicate with the monitor-sequencer. The OF module is described in the Sec. 2.2.2.

Three 240x180-pixel DVS event slices are stored in RAM. These slices are like binary image frames from conventional cameras but in the case of DVS we can arbitrarily select the slice interval d. One is the current slice starting at time t and the other two are the past two slices starting at times t-d and t-$2d$. At intervals of d, the rotation control logic circulates the three slices. The t slice accumulates new data. It starts out empty and gradually accumulates events, so it cannot be used for matching with past slices. The two past slices are used for OF, but the OF computation is done at the location of each event stored into the t slice, and thus is driven by the incoming events. Slices are stored in block RAM on the FPGA. The total size of the RAM is 240x180x3, matching the DVS pixel array size. It is generated by the IP Core of Xilinx.

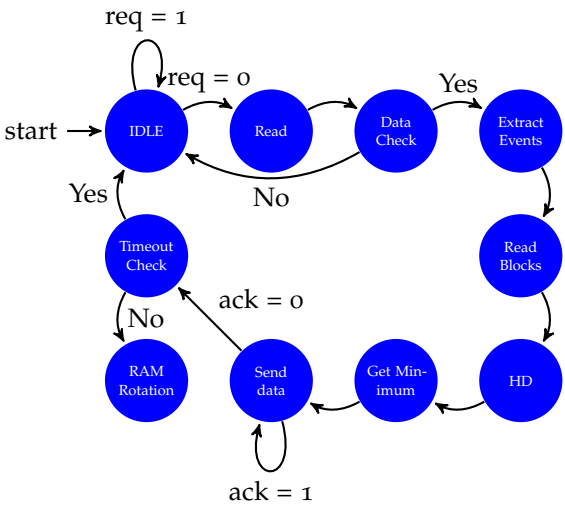


FIGURE 2.2: Finite state machine

2.2.2 *Optical flow algorithm*

The current algorithm's search radius is 1 and thus the search range is [-1, 1] pixel. Therefore, for every reference block, we compare it with 9 candidate blocks. When an event arrives, a single reference block from slice*t-d* and When an event arrives, a single reference block from slice *t-d* and 9 blocks from slice *t-2d* are sent to the HD module to calculate the distances. In the current implementation, the block contains 9x9 pixels. For the *t-d* slice, we use only one center block as the reference. The algorithm finds the most similar block on the *t-2d* slice. According to the brightness-constancy assumption of OF, we should see a similar block in the *t-2d* slice for the block that best matches the actual OF. We search over 8 blocks centered on the 8 neighbors of the current event address and one block centered on the reference and choose the one with minimum distance.

2.2.2.1 *Hamming Distance*

The implementation of one HD block is shown in Fig 2.3. A total of 81 XOR logic gates receive input from the corresponding pixels on the slices. The XOR outputs are summed to compute the HD.

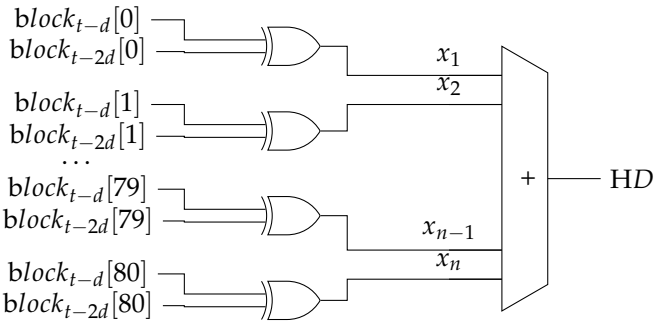


FIGURE 2.3: Hamming Distance implementation for one 9x9 block match. There are 9 of these circuits for the 9 flow directions.

2.2.2.2 *Minimum Distance Computation*

The last step of the algorithm is to find the minimum distance candidate. Part of the novel minimum circuit is shown in Fig 2.4. It is a parallel implementation that outputs the index of the minimum distance direction. For instance, if we need to find the minimum among 5 data: HD0-4 (output from Fig 2.3), the circuit can be divided into 5 parts. The first part in Fig 2.4 compares HD0 with all other data and outputs a count of how many times data0 is larger than HD1-4. The other 4 parts are implemented in the same way and all those parts are computed concurrently. At the end, the part whose sum is zero is the minimum candidate. Thus, the minimum distance candidate is determined in one clock cycle.

2.3 EXPERIMENTAL RESULTS

We used the Opal Kelly XEM6310MT development board as our to implement our algorithm. This board featured a Xilinx Spartan 6 family chip xc6slx150t. It has 184304 Flip-Flops and 92152 LookUp Tables (LUT) and 4MB block memory. The implemented OF design occupies 0.9% of the Flip-Flops, 5% of the LUTs and 5% of the block RAM. For the test dataset, we use the event-based optical flow benchmark dataset in [115] which also provides the evaluation method and the ground truth.

We tested three sample real DVS recording datasets: the *boxes*, *pavement*, and *gravel* corresponding to edge, sparse points, and dense texture respectively. The *boxes* scene has a box in the foreground and clutter in the background and the camera pans to the left, producing rightwards global trans-

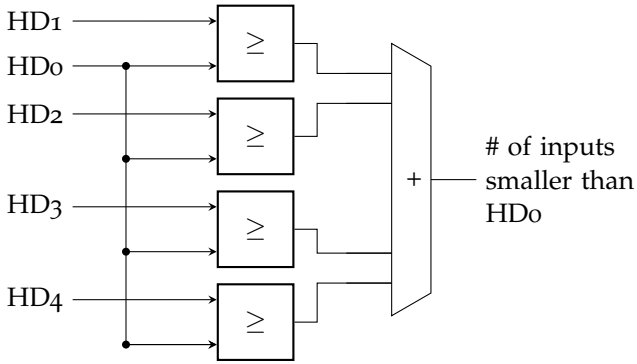
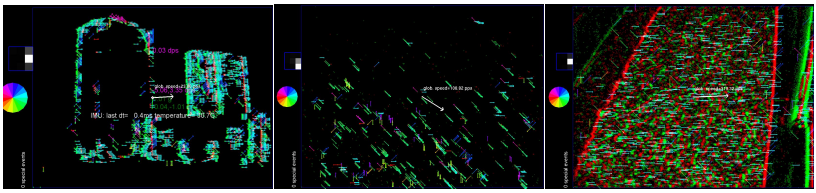


FIGURE 2.4: Sort algorithm implementation block for HD0, simplified for 5 inputs rather than 9. There are 9 of these blocks.



(a) Boxes translation (b) Pavement on grass (c) Gravel

FIGURE 2.5: OF Results. The arrows are the flow vectors and their length represents the speed (determined by the slice duration d). DVS On events are green and Off events are red. The color wheel indicates the flow vector direction color. The 2D gray scale histogram (zoom in to see it clear) above each color wheel shows the distribution of flow event directions (here we use 9 direction bins) in the time slice. The brightest bin votes the highly possible direction of the global motion. (a) is the boxes scene from [115] with $d = 40$ms. (b) is pavement recorded by a down-looking DVS; $d = 10$ms. (c) is a gravel area with $d = 3$ms. For clarity, event rate down sampling was used to compute 1 flow event for every 100 DVS events.

lation mostly of extended edges. In the pavement dataset, the camera was down-looking and carried by hand; the flow points downwards and to the right. Imperfections in the pavement cause sparse features. The *gravel* dataset is recorded outside and has dense texture; movement is eastward.

AAE	transBoxes
PM_{hd}	42.68±33.82
LK_{sg}	**30.30±44.35**
LK_{bd}	98.92±42.24
LP_{orig}	77.18±33.73
LP_{sg}	47.52±54.44

(a) AAE comparison

AEE	transBoxes
PM_{hd}	**17.86±6.31**
LK_{sg}	24.72±26.11
LK_{bd}	37.00±15.18
LP_{orig}	93.02±107.02
LP_{sg}	98.32±82.5

(b) AEE comparison

TABLE 2.1: OF algorithm's accuracy

The block-matching OF results are shown in Fig 3.11. It can be seen that in each scene, most vectors point correctly east for box translation, southeast for the pavement scene, and east for the gravel scene. Errors are mostly caused by DVS noise or aperture ambiguity at the extended edges.

2.3.1 *Accuracy analysis*

[115] proposed two ways to calculate event-based OF accuracy, based on similar metrics used for conventional OF. One is called Average Endpoint Error (**AEE**) and the other is Average Angular Error (**AAE**). AAE measures error in the direction of estimated flow and AEE includes speed error. These two methods are already implemented in jAER [139]. They use IMU data from a pure camera rotation along with the lens focal length as the ground truth. Since the output data of the sequencer lacks IMU data, we measured the OF accuracy using the PC implementation. The algorithm pipeline between FPGA and PC is identical, so it will not influence the accuracy. The result is also compared with [115]. We chose two variants of the EBLK and LP algorithms. The errors from all algorithms are shown in Table 2.1. PM_{hd} represents the block matching algorithm with HD metric.

As shown in Table 2.1, the block matching algorithm has the best accuracy for AEE and second-best for AAE, partly from an appropriate choice of the sample rate that matches the dataset motion.

Fig 2.6 shows the relationship between the block radius and AAE. It indicates that bigger block dimension leads to better accuracy. However, larger blocks consume more logic and reduce the spatial resolution of the flow.

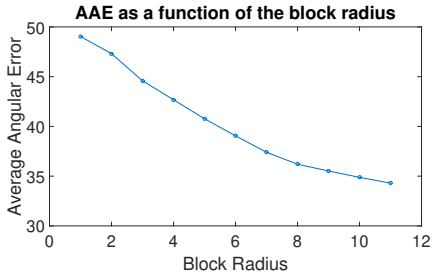


FIGURE 2.6: The relationship between the block radius and AAE [1].

The comparison between the PC and FPGA implementation complexity is discussed next, in 2.3.2.

2.3.2 *Time complexity analysis*

The time complexity of the software grows quadratically with the block size while only linearly in FPGA. The processing time of the algorithm contains three parts: reading data from three slices, HD calculation and looking for the minimum. Both FPGA implementation and software implementation on PC consume linear time to read data from RAM since multiple data cannot be read from one RAM simultaneously. However, the latter two parts take constant time (2 clock cycles) on FPGA while quadratic time on PC. In summary, the processing time on FPGA is (block dimension + 2) cycles. In this work, FPGA runs at 50MHz frequency and the block dimension is 9. Thus, the whole algorithm will take only 220ns per event, i.e. 0.22us. On PC, it takes 4.5us per event for (admittedly non-optimized) jAER to run the algorithm. The implementation on FPGA is 20 times faster than that on the PC.

2.4 SUMMARY

This chapter presented a block matching method to estimate the event-based OF on FPGA in real time. The software computational cost of HD increases quadratically as the block size increases, however, in FPGA, all bits in the block can be calculated at the same time which leads to a constant time for all block sizes. This greatly reduces the overall computation time

1 Tested on boxes_translation scene from [115].

for the FPGA implementation, which is 20 times faster than the software implementation. In the final implementation, every single incoming event is processed (allowing an input event rate of up to 5 Meps to be handled using a modest FPGA clock of only 50 MHz). However, processing every event is not required, as illustrated in Fig. 3.11(d), where OF computation is downsampled, but the DVS events still indicate locations to estimate the flow.

There are three possible improvements. The current implementation estimates only the direction of flow and not speed. Measuring speed requires additional search distances and there are well-known algorithms for efficient search [140]. Secondly, other distance metrics should be explored because event sequences collected onto the slices usually have different length due to noise and HD is somewhat ambiguous [141]. Finally, it is worth implementing feedback control on the slice duration to better exploit the unique feature of DVS event output that it can be processed at any desired sample rate. This capability is a key distinguishing characteristic from frame-based vision, where the sample rate and processing rate are inextricably coupled. It could allow a block-matching approach for DVS that achieves high OF accuracy even with only small search distances and modest hardware resources. The next chapter explores these improvements.

3

ADAPTIVE BLOCK MATCHING OPTICAL FLOW FOR EVENT-BASED CAMERA¶

3.1 INTRODUCTION

In Chapter 2, we showed a basic BMOF for event-based camera. The block size was 9x9, and the search radius was only one pixel. This small radius and block size severely limited OF accuracy. We thus improved BMOF in three aspects:

- We use Sum of Absolute Differences (**SAD**) as the metric to improve the search accuracy and Diamond Search (**DS**) to increase the search efficiency.
- We proposed a new event slice accumulation method to make the algorithm more adaptive to the dynamic environment.
- We used a feedback signal based on the result OF histogram to fine-tune the slice duration further.

The novel algorithm is called Adaptive Block Matching Optical Flow (**ABMOF**). This chapter is organised as follows: Sec.3.2 describes the algorithm in detail. Sec.3.3 shows the experimental result and Sec.3.4 concludes this chapter.

3.2 ABMOF ALGORITHM

The pipeline of ABMOF is summarized in Fig 3.1. When a new event arrives, the event's timestamp is used by the rotation logic to determine whether the event slice is to be rotated. If yes, the slices are rotated and the slice duration d or event count parameter K or k is adapted based on the current slices's OF distribution. The adapted slice duration is sent as an input to the rotation logic. The adaptation takes the OF distribution of the previous slice as the input. We use a dashed connection in the figure to represent their relationship. Details of the rotation logic are introduced in Sec. 3.2.2.

¶ A substantial content of this chapter is published in [142].

Symbol	Description	Typical values (default)
$w \times h$	width × height of pixel array	346x260
d	slice duration	1–100 ms (50)
K	global event number	1k–50k events (10k)
k	area event number	100–1k events (1k)
a	area dimension subsampling	5 bits
b	block dimension	11–21 pixels (21)
r	search radius	4–12 pixels (4)
s	# scales	1–3 (2)
p	skip count on PC for real-time	30–1000
g	# bits for slice counts	1–7 (3)
g	# bits for slice counts	1–7 (3)
D	average match distance	ideally $r/2$
(v_x, v_y)	OF result	pixels/sec (pps)

TABLE 3.1: Symbols, description, and typical values/units.

All the new events will be accumulated to multi-scale slices. If the system is busy, the OF calculation for the event is skipped. Otherwise, it triggers the OF calculation. The event skipping mechanism is introduced in Sec. 3.2.5. After removing events with too sparse blocks, the OF histogram is updated.

All the parameters that are used in this paper are summarized in Table 3.1.

3.2.1 *Block-Matching DVS time slices*

To make the chapter self-explanatory, we first make a brief wrap-up of BMOF in this subsection. Fig. 3.2 shows the main principle of BMOF. Three event-slice memories store the events as 2D event histograms: Slice t accumulates the current events. Slices t-d and t-$2d$ hold the previous two slices. d is the slice duration. When a new event arrives, it is accumulated to slice t by either incrementing the pixel value, or adding the polarity of the event to it. Which of these is done depends on whether we ignore the event po-

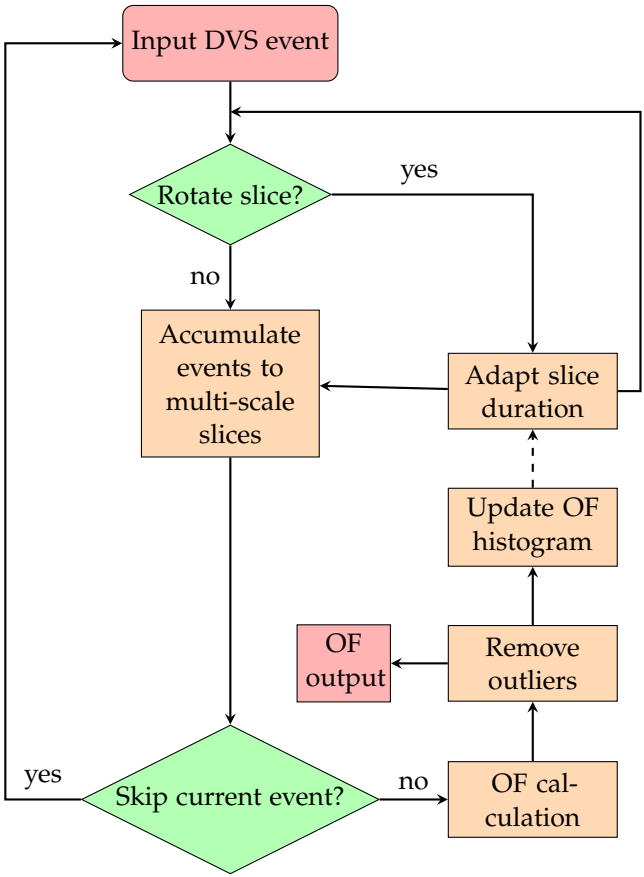


FIGURE 3.1: The pipeline of our algorithm

larity. For the experiments in this work, we usually ignored event polarity because the accuracy did not change significantly even if it is included. Including polarity may enable better block matching, but it carries the price that one bit of the pixel memory is used for the sign bit. After accumulating the event, then the other two slices are then used to compute the OF based on the current event's location. When multi-scale slices are used (Sec. 3.2.4), then each slice is a pyramid of s slices.

A reference block ($b * b$ pixels) is centered on the incoming event's location on the *t-d* slice map (red box in slice *t-d*). The best matching block on the *t-2d* slice is found based on SAD inside a $(2r + 1)^2$ search region,

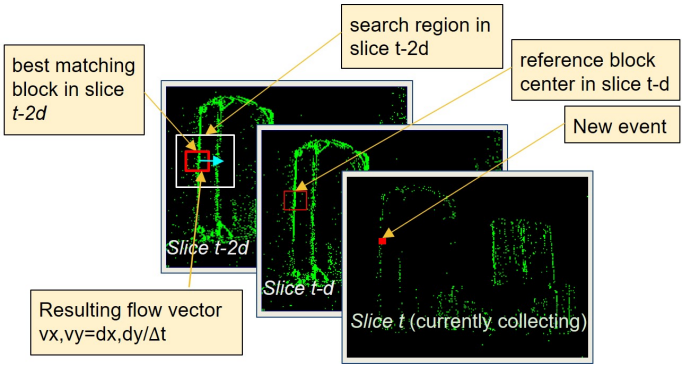


FIGURE 3.2: BMOF block matching, on `boxes` from [115]

shown as a white rectangle in the *t-2d* slice. Thus, the optical flow result is obtained by using these two blocks' offset (d_x, d_y), divided by the time interval Δt between these two slices. The time of each slice is taken as the average of the first and last timestamp of events accumulated to each slice.

The slices are rotated according to the slice rotation logic (Sec. 3.2.2). The rotation discards the *t-2d* slice and uses its memory for the new slice *t*; similarly slice *t* becomes slice *t-d* and slice *t-d* becomes slice *t-2d*. In BMOF, the slice duration *d* was set by user manually. It is not convenient for general application since it limits the speed range. In this work, we propose several methods to adjust it adaptively.

3.2.2 *Slice rotation methods*

Slice rotation is the core part of our algorithm. It calculates when to rotate the slices to ensure good slice quality. Good slices should have sharp features, not too much displacement, and not be too sparse. This goal is achieved by feed-forward and feedback control. We show the details of these two algorithms in the following subsections.

3.2.2.1 *Feedforward slice rotation*

The new events are accumulated into the latest slice, slice *t*. Slice *t* is only used for accumulation. After that, it will be rotated to be as a past slice and used for OF calculation.

In the original BMOF work, we implemented the method *ConstantDuration*, where each slice has the same duration d. Another obvious method is to rotate slices after a constant number K of events have been accumulated, called *ConstantEventNumber*.

- *ConstantDuration*: Here, the slices are accumulated to time slices uniformly with duration d. This method is what we reported before and corresponds most closely to conventional frame-based methods. It has the disadvantage that if the scene motion is too fast, then the movement between slices may be too large to be matched using a specified search distance. If the movement is too slow, then the features may not move enough between slices, resulting in reduced flow speed and angle resolution.

- *ConstantEventNumber*: Here, the slices are accumulated until they contain a fixed total count of DVS events K. If K is large then the slices will tend to have larger d. But if the scene moves faster, then the rate of DVS events also increases, which for fixed K will decrease d. Therefore the *ConstantEventNumber* method automatically adapts d to the average overall scene dynamics.

A drawback of the *ConstantEventNumber* method is its global nature. For scenes which have lots of or very few textures, it is impossible to set a suitable global K. In order to address this problem, we propose a new rotation method called *AreaEventNumber*.

- *AreaEventNumber*: Instead of rotating the slices based on the sum of the whole slice event number, *AreaEventNumber* will trigger the slice rotation once any one of the area's event number (Area Event Counters) exceeds the threshold value k. Each area is $(w \times h)/2^a$ pixels, *i.e.*, 10×8 pixels.

$$
\begin{aligned}
e_i &= \{x, y, t_s, pol\} \\
I_t &= \{e_0, e_1, \cdots, e_O\} \\
k_{mn} = \sum_{i=1}^{O} f_{mn}(e_i) \qquad &\text{where } f_{mn} = \begin{cases} 1 & \lceil \frac{x}{2^a} \rceil = m, \lceil \frac{y}{2^a} \rceil = n \\ 0 & \text{otherwise} \end{cases}
\end{aligned} \tag{3.1}
$$

$$
ConstantNumber: \ O >= K \tag{3.2}
$$

$$AreaEventNumber: \ \exists k_{mn} >= k \tag{3.3}$$

The definitions of *ConstantEventNumber* and *AreaEventNumber* can be represented by math equations. As shown in the equation 3.1, the current event slice I_t is divided into $m \times n$ small areas. If the total event number O is bigger than the threshold K, the slice is rotated. This method is called *ConstantEventNumber* (eqaution 3.2). If any event number of the small area $k_m n$ exceed the area event number threshold k, the slice is rotated. This method is called *AreaEventNumber* (eqaution 3.3).

By using the *AreaEventNumber* method, slice rotation is data-driven by the accumulation of DVS events, but adapts the slice durations to match the area of the scene which has the most DVS activity. This adaptation prevents under-sampling that causes displacement that is too large to match between slices. Compared with *ConstantEventNumber* method, it preserves the advantage that the generated slices adapt to the scene dynamics. The local adaptability makes the slices more robust to variation and distribution of scene texture.

To make it even more robust and adaptive, the slice event number k is also adaptive to the scene. When the scene moves fast, the parameter k will be increased. Otherwise, it is decreased. Adaptation of k is further described in Sec. 3.2.2.2.

An example to demonstrate these three methods is shown in Fig 3.3. The blue arrows pointing to the three time axes represent these three rotation method results. It is obvious that both the time interval and the event number interval are fixed for *ConstantDuration* and *ConstantEventNumber*. However, both of them vary in the *AreaEventNumber* method which makes it more adaptive to the dynamic scene.

In Fig 3.4, we compare these three methods on two different scenes, one has sparse textures and the other has dense textures. Among them, Figs. 3.4(a), 3.4(c), and 3.4(e) are obtained from the same dense scene by the three different methods. Figs. 3.4(b), 3.4(d), and 3.4(f) are obtained from a sparse scene. Both dense and sparse scenes use the same parameter for every method which is $d = 10ms$ for *ConstantDuration*, $K = 10000$ for *ConstantEventNumber* and $k = 700$ for *AreaEventNumber*. The resulting slice durations are shown overlaid on each scene.

Neither *ConstantDuration* nor *ConstantEventNumber* work well on both of these two scenes with fixed values of d or K. For example, *ConstantDuration* fails in the sparse scene because d was set for the faster motion in Fig. 3.4a and the duration was too short for the slower motion in Fig. 3.4b.

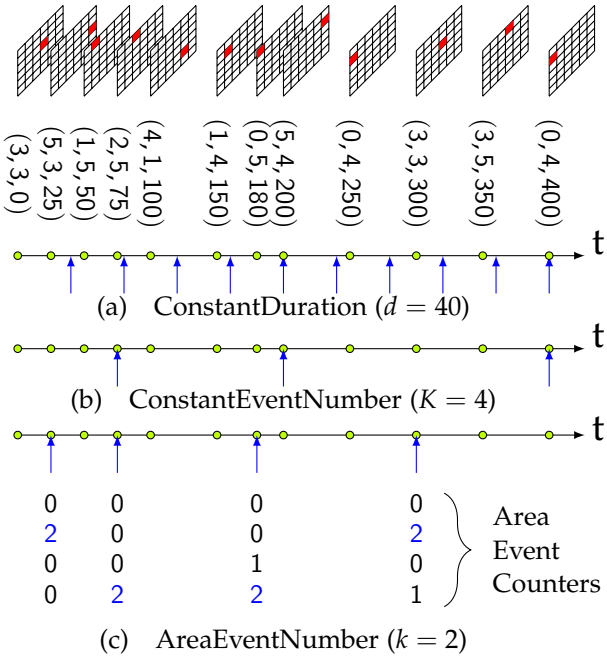
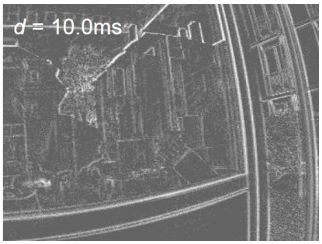


FIGURE 3.3: Three feedforward slice rotation methods. The event stream is at the top of the figure. The information including event address (x,y) and timestamp is shown under the event stream. An example of these three slice methods is demonstrated here, with (a) slice duration $d = 40$, (b) global event number $K = 4$ and (c) area event number $k = 2$.

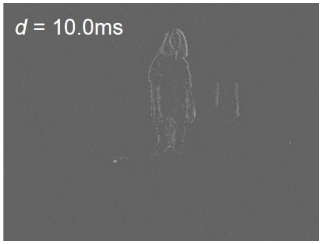
ConstantEventNumber makes the slice too short in duration in the dense scene in Fig. 3.4c, because K was set to make a good slice for Fig. 3.4d. However, *AreaEventNumber* with fixed parameter k functions well on both of scenes, because it correctly creates the Fig. 3.4f slice after being set for the dense scene in Fig. 3.4e. It shows that *AreaEventNumber* is more robust to dynamic scene content.

3.2.2.2 *Feedback Control of Slice Duration*

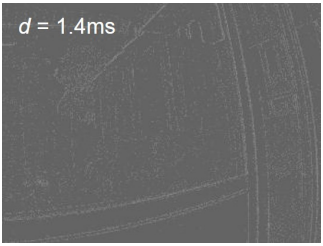
Another method to automatically adjust the slice duration is possible via feedback control. An optical flow distribution histogram is reset after each slice rotation and then collects the distribution of OF results. The histogram's average match distance D is calculated. If $D > r/2$ where r is



(a) *ConstantDuration* on dense texture scene

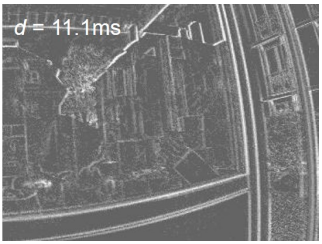


(b) *ConstantDuration* on sparse texture scene

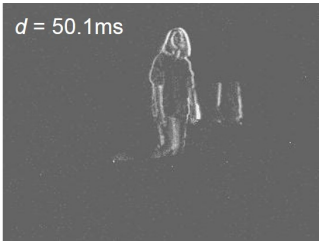


(c) *ConstantEventNumber* on dense texture scene

(d) *ConstantEventNumber* on sparse texture scene



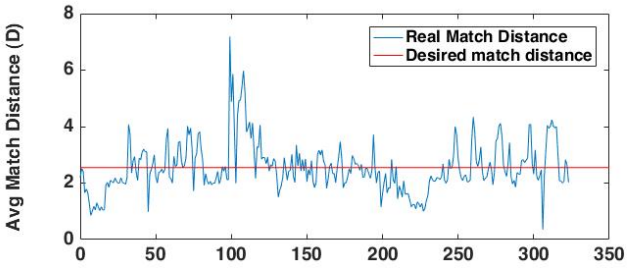
(e) *AreaEventNumber* on dense texture scene



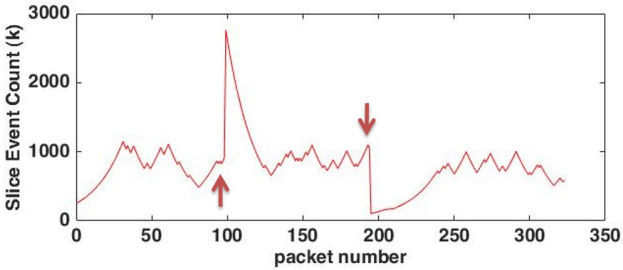
(f) *AreaEventNumber* on sparse texture scene with

FIGURE 3.4: Comparison between event slices generated by three methods.

the search radius, it means that the slice duration is too long, and so the slice duration or event number is decreased. Otherwise, if $D < r/2$, then it indicates the slices are too brief in duration, and the slice duration or event number is increased. It is possible that slice durations that are too brief or lengthy result in OF results of very small matching distance that are the result of a bias in the search algorithm towards zero motion (small match distance). Stability is improved by limiting the slice duration range within application-specific limits. For the control policy, we so far used bang-bang control. A fixed factor of $\pm 5\%$ adjusts the slice duration, where the sign of the relative change of duration is the sign of $r/2 - D$. More sophisticated control policies are clearly possible, since the value of the error is directly predictive of the necessary change in the duration.



(a) OF result's match distance



(b) Feedback on slice event count

FIGURE 3.5: Feedback on slice event number. (a) shows the OF result's real match distance and its desired match distance. (b) represents the slice event count number.

Since the principle of feedback control on event number and slice duration is similar, we show only an example of feedback control on event number k here. The data in Fig. 3.5 shows an example of event number control using the *AreaEventNumber* rotation policy with a feedback control of k. Fig. 3.5(a) shows the average OF match distance D. The feedback control of the event number holds D at its $r/2$ value of about 2.5 pixels. Fig. 3.5(b) shows the event number k. It has a steady state value of about 1000. Around packet 100 (1st arrow), k was manually perturbed to a large value, resulting in a increase in D, but it rapidly returns to the steady state value. At around packet 200 (2nd arrow), k was manually reduced to a small value, resulting in a small D of about 1 pixel. Again, D returns to the steady state value. This data shows the stability of the event number control with this feedback mechanism.

3.2.3 *Search method*

The implementation of BMOF searched only the target block and its 8 nearest neighbors. An improvement is offered by extending the search range to a larger distance range r. The block matching search method can be done by exhaustive full search. It has the best search accuracy but is expensive since the cost grows quadratically with r. A more efficient method is diamond search [143], which we implemented. It makes a trade-off between computation and accuracy. Our results show that it has about 90% chance to hit on the best matching block with a cost 14X less than the full search, for $r = 12$. Using the diamond search improves the algorithm's real-time performance significantly.

3.2.4 *Multi-scale and multi-bit event slices*

A limitation of the approach described so far is the limited dynamic speed range of the method, since matching can only cover a spatial range of square radius r around the reference location. One way to increase the search range by a factor of 2^s with only s linear increase in search time is to use a multi-scale pyramid [144]. In this method, events are accumulated into a stack of time slices. Each slice in the stack subsamples the original event addresses in x and y directions by a factor of 2 more than the previous scale. I.e., if $s = 0$ means the original full resolution scale, then events are accumulated into scale s by first right shifting the event x and y addresses by s bits, and then accumulating the resulting event into the

scale s slice, which has only $1/2^s$ as many pixels for each dimension. For example, in the $s = 1$ scale slice, each pixel accumulates events from a 2x2 pixel region in the full resolution original pixel address space. To prevent saturation, we use multiple bits g for each value; for example $g = 3$ allows up to 7 unsigned events for each pixel when we ignore the event polarity, or up to ± 2 events when we use polarity. Thus, the total slice memory required for an N pixel sensor is $3Ng\sum_{m=0}^{s-1} 2^{-m}$ bits.

To compute the OF, each event is processed independently for each scale. The match that has the minimum SAD is selected as the OF. Using multiple scales is beneficial particularly in noisy situations, where the event flow is sparse. The binning of events helps to find good matches.

3.2.5 *Adaptive event skipping*

For high speed or densely textured scenes, the event rate becomes high. If we still compute OF for each event the real-time performance will be influenced dramatically and the algorithm quickly falls behind the actual incoming event rate. To address this problem in software, we propose an event skipping method. If the processing time is higher than a threshold we set, the following events do not have their OF calculated. However, they are still accumulated to the current slice. By doing this, we can get a trade-off between the OF density and the real-time performance. The adaptive event skipping algorithm uses a skip parameter p, which is increased or decreased depending on the frame rate. If the actual frame rate is slower than the desired frame rate set by the user, it means it takes too much time to process the event, then p increases. Otherwise, it decreases. On a Corei7-975 PC in Java 1.8, the ABMOF implementation requires about 15 us per event with the default parameters in Table 3.1 and $p = 1$. With $p = 1000$, the time drops to an average of about 260 ns/event; there is some overhead for each packet and for slice rotation, which is why it only drops by only a factor of 600X. For hardware ABMOF implementation on FPGA, which we will show in Chapter 6, it is fast enough to process every event in most cases. Even if in the extreme case, it is also easy to implement this mechanism on hardware. A FIFO forms a buffer for the incoming events. The event skipping will be designed as a switch and will be connected to the FIFO half-full flag. If the FIFO is half-full, it means the processing time is falling behind, and event skipping will be enabled.

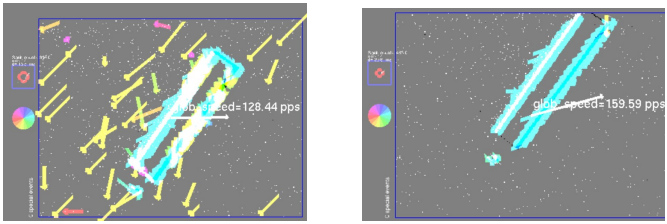
3.2.6 *Sparsity checking*

To improve the accuracy, we developed sparsity checking to filter out OF results with poor matching quality. We use two parameters to reject the blocks that are too sparse. One parameter is `validPixOccupancy`; it determines the percentage of valid pixels in two blocks that will be compared. Valid pixels are the pixels where events were accumulated. The reason for setting this parameter is sometimes the blocks are too sparse, which makes the distance metric get a meaningless result. By only calculating matching for blocks that are filled with sufficient valid pixels, we can reject misleading results.

A second parameter is called `maxAllowedSadDistance`. The minimum distance between the reference block and the candidate block must be smaller than `maxAllowedSadDistance`, otherwise the OF event will be rejected. Thus, the best matching search block may actually be a poor match, and `maxAllowedSadDistance` allows rejecting the best matches if the match distance is too large.

The effect of these parameters is shown in example data in Fig 3.6 from a simple case of a black bar moving up and to the the right. The flow results are visibly cleaner using these outlier rejection criteria.

Both mechanisms are easily implemented in hardware. For example, the valid pixel occupancy can be realized by pixel subtraction units that output a large value if both operands are zero. The confidence threshold can be realized by a comparator on the final best match output result that flags a distance that is too large.



(a) Without outliers rejection (b) With outliers rejection

FIGURE 3.6: Example of outlier rejection using `maxAllowedSadDistance` and `validPixOccupancy`. (a): without outlier rejection. (b): using outlier rejection with `maxAllowedSadDistance=0.5` and `validPixOccupancy=0.01`.

3.3 EXPERIMENTAL RESULTS

In this section, we first describe the dataset we recorded for the experiment in 3.3.1. It is called ABMOF18.

3.3.1 *ABMOF18 dataset*

ABMOF18 is a DAVIS optical flow (OF) dataset which has a variety of complicated real scenes. It is recorded using a latest DAVIS camera DAVIS346 which has 346x260 pixel resolution and integrated on-chip APS readout circuits, allowing a maximum APS frame rate of about 50Hz. This DAVIS346 has pixels with integrated microlenses, optimized photodiodes, and antireflection coating, which together increase the effective quantum efficiency to about 24% compared with the previous DAVIS240C QE of 7%. It also includes its own IMU that measures camera rotation and acceleration. For movement with only camera rotation, the IMU can be used as a method to obtain the ground truth optical flow. The ABMOF algorithm is implemented in jAER as the event filter PatchMatchFlow and can be explored and compared with other algorithms; see the previous benchmarking dataset DVSFLOW16 [115]. ABMOF18 dataset could be downloaded using a tool called Resilio sync. The link is provided in the Appendix Sec. A.7.

3.3.1.1 *recordings*

We provide two types of data. The first type of data is recorded originally using jAER. It consists of two datasets: gravel, and office running. Gravel is recorded outdoor in which the camera is downwarding to the gravel and the camera is rotated around a point in the scene to cause the scene to both rotate and translate. Both rotation and translation movement are recorded in this dataset. Office running is an indoor dataset. It is recorded by a person holding the DAVIS346 running through our laboratory. Both of these datasets provide IMU data. This is provided for IMU groundtruth, when it is possible for pure camera rotation. For details about event-based IMU optical flow (OF) ground truth, see [115].

The second type of data is originally from the Robotics and Perception Group at the Univ. of Zurich. For the details of this dataset, refer to `http://rpg.ifi.uzh.ch/davis_data.html`. The format of this dataset is in rosbag. We convert it to .aedat format using a conversion tool. The converted files

are provided here. They contain only DVS data; IMU and APS frames are not converted.

3.3.1.2 *Matlab scripts*

The Matlab scripts provide a method to convert OF global translation result to VO for dataset which has known groundtruth. The main file is plotGlobalVelocity.m. In Matlab, first enter the folder containing the script, and then simply run plotGlobalVelocity. When it's started, it will ask the user to select a folder that needs to be converted.

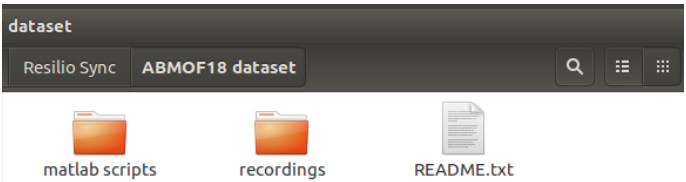


FIGURE 3.7: Folder structure of the matlab scripts folder.

Calib.txt and groundtruth.txt are provided by the original dataset. Users should provide the depth in depth_properties.m and OF global translation result file.

The software implementation of the algorithm is open source. It is called PatchMatchFlow [145] in jAER [146]. In this section, we show several experiments to validate the ABMOF algorithm. They can be classified into two types. Type I are the experiments with ground truth OF using two datasets. The first dataset `slider_hdr_far` is from [1]. It shows a flat poster scene with a fixed depth of 58 cm where the camera is moved laterally by a motorized cart, resulting in uniform flow of about 90 pixels per second (pps). It was recorded with high lighting contrast. The second dataset `pavement_fast` is a scene with extremely fast flow of 34k pps recording from a car, with a down-looking camera recording an asphalt pavement.

The files converted to jAER aedat format are included in ABMOF18 dataset. We show both the qualitative and quantitative results in these experiments; see Sec. 3.3.2

In type II experiment, we test our algorithm on several complex scenes. They consist of camera rotation over gravel (`gravel`), flow through an indoor office environment (`office`), and uniform flow created by a variety of shapes (`shapes`, from [1]). The dataset is provided[1] to support the tests for other future algorithms. Due to the unknown ground truth in these files, we show only a comparison between the ABMOF and Lukas Kanade results using the generated slices for these data; see Sec. 3.3.3.

For the `slider_hdr_far` data, the groundtruth camera position is provided for each time point and the scene has provided uniform depth. By using the camera calibration data and pinhole camera model, we converted the pose groundtruth data to a global optical flow groundtruth.

For `pavement_fast`, we manually measured the flow using a jAER [146] software filter called *Speedometer*, which allows using the mouse to mark a moving feature point at different time points and measures the distance and time between these marks.

3.3.2 *Type I experiment result*

We show the results of type I experiments in this subsection. We measured four metrics to evaluate the algorithms. They are Event Density (ED), translational Global Flow (GF), AEE and AAE. ED is the fraction of DVS events that result in OF results. DVS events are skipped because block matching fails to pass the density checking tests; we set $p = 1$ for these tests. ED relates to the density of the flow computation. LK has very low density because it relies on features. An ED of 100% means that all pixel brightness changes result in OF events. AAE and AEE are defined for DVS OF in [115].

Besides the ABMOF, we also implemented the LK OF calculation based on our generated adaptive event slices using OpenCV and we call it ABMOF_LK. ABMOF_LK uses our algorithm to set the time slice duration, and these generated slices are treated as conventional gray scale image frames. In ABMOF_LK, corners are first extracted by Shi-Tomasi corner detector [147] and then they are are passed to the LK tracking algorithm implemented in OpenCV [148]. LK estimates the OF result based on these features.

1 http://tiny.cc/itpauz

We also compared the AMBOF OF methods with previously published implementations from [115]: *DirectionSelectiveFlow* (DS) [117], the event-based *LucasKanadeKFlow* (EBLK) [118], and *LocalPlanesFlow* (LP) [120].

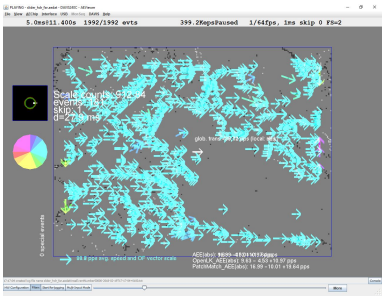
3.3.2.1 `slider` *scene*

Fig. 3.8 shows the qualitative results of ABMOF and ABMOF_LK on the `slider_hdr_far` data. This is a high dynamic range scene of a flat poster with uniform flow about about 90 pps. Because of the lighting contrast, the APS images are sometimes extremely over- or underexposed, but the DVS events respond to local brightness changes. Table 3.2 and Table 3.2 report the quantitative comparison. By the VO groundtruth to OF groundtruth conversion, we can compare them over time, as shown in Fig. 3.9. Table 3.2 shows that ABMOF_LK's GF error on the `slider_hdr_far` data is less than 1pps and ABMOF's GF error is less than 4pps. ABMOF_LK is more accurate than ABMOF, but has much lower ED. Fig. 3.9(a) shows a very clear periodic oscillation in v_x for both *ABMOF* methods, which is caused by the simple bang-bang control of the slice duration coupled with match distance quantization. This oscillation is confirmed by the trace *ABMOF fixed with 45ms*, where we fixed $d = 45$ms; its v_x flow is a bit too small because of the quantization of the match distance. The conventional LK method on the frames also obtains the average correct flow (and does not have the controller oscillation), but as seen in Fig. 3.8(c), this estimate is sometimes based on a single keypoint. That is the cause the outliers for *Frame_based LK* in Fig. 3.9(b) around 5s and 6s where the frame LK method suffered large aperture error.

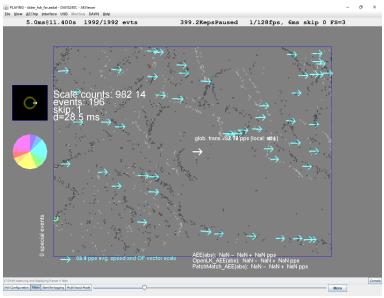
This experiment validates that the slice rotation methods result in quantitative flow magnitude that is the same as from frame-based LK . The ABMOF methods are oscillatory using the current k controller, but have much higher density than the frame-based LK method. All ABMOF methods are all much more accurate and less noisy than the prior DS, EBLK, and LP methods.

3.3.2.2 `pavement_fast` *scene*

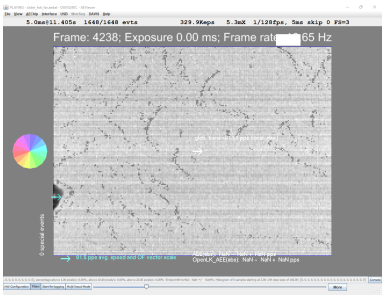
Fig. 3.10 shows the results of a very high speed experiment on `pavement_fast`, which was recorded from a car with a down-looking camera aimed at the asphalt pavement road surface. The global flow is an extremely fast 32k pps, which means that a pixel crosses the 346-pixel array in about 10 ms. Figs. 3.10(a) - 3.10(b) compares ABMOF and ABMOF_LK on DVS



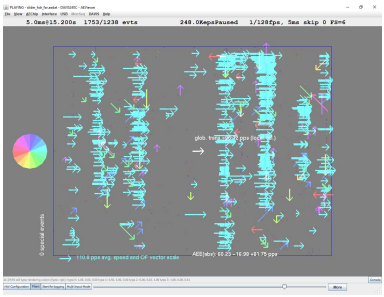
(a) ABMOF



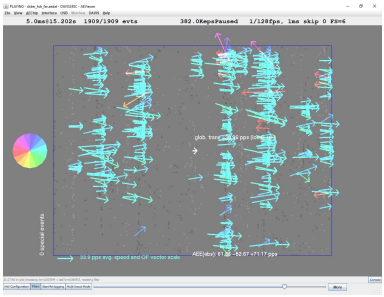
(b) ABMOF_LK



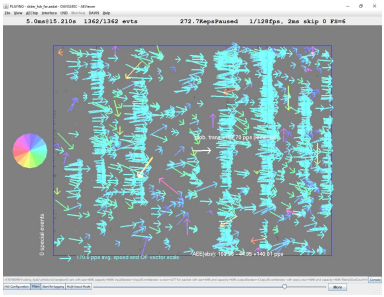
(c) APS image 2 and LK result



(d) DS

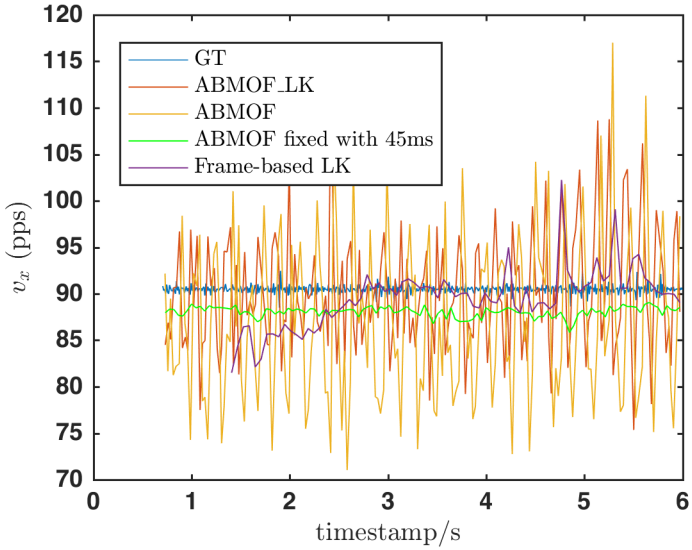


(e) EBLK

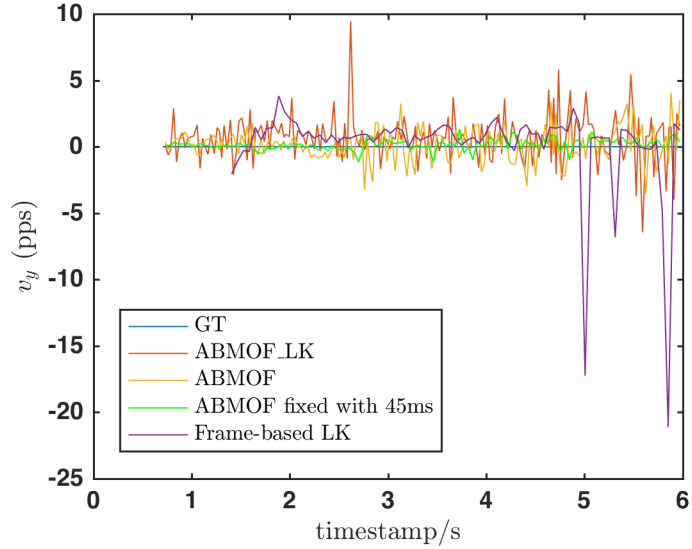


(f) LP

FIGURE 3.8: Result of ABMOF, ABMOF_LK and standard LK on image frames on `slider_hdr_fast`. For 3.8(a) and 3.8(b), we use *AreaEventNumber* with feedback enabled, $r = 4$, and $s = 2$, and used standard LK on successive APS images in 3.8(c).



(a) v_x for `slider_hdr_far`



(b) v_y for `slider_hdr_far`

FIGURE 3.9: Comparison of measured and ground truth flow between OF methods on `slider_hdr_far`

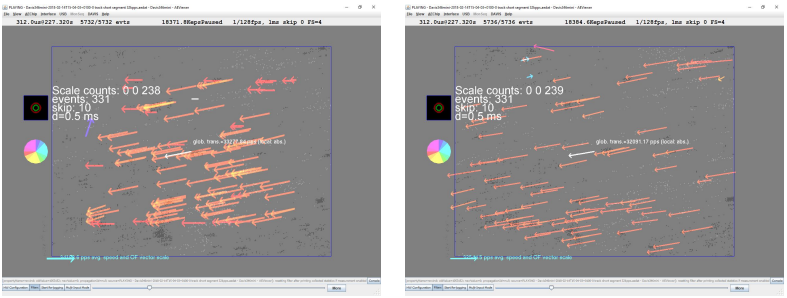
Method	Event density	Global flow (pps)	AEE (pps)	AAE (°)
Groundtruth	-	$[90.50, 0] \pm [0.43, 0]$	-	-
ABMOF_LK	0.39%	$[89.75, 0.44] \pm [6.30, 3.56]$	8.75±27.51	2.95±3.41
ABMOF	37.96%	$[86.85, 0.17] \pm [8.46, 1.25]$	12.68±16.28	3.66±8.31
Frame_based_LK	-	$[89.51, 0.20] \pm [3.203.48]$	5.47±42.07	1.30±4.72
DS ([115, 117])	49.86%	$[74.97, 2.98] \pm [17.42, 4.79]$	57.71±53.31	21.46±39.13
EBLK ([115, 118])	17.53%	$[28.06, -0.11] \pm [4.09, 1.32]$	60.32±15.92	13.52±25.51
LP ([115, 120])	83.88%	$[161.14, 11.69] \pm [8.67, 12.13]$	99.00±75.86	16.99±24.41

TABLE 3.2: Comparison of algorithm's overall accuracy on `slider_hdr_far`.

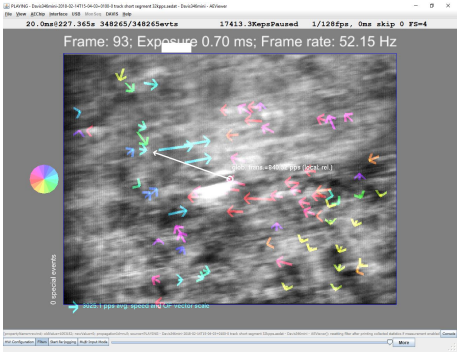
time slices, and Fig. 3.10(c) shows conventional LK on successive DAVIS APS images (the 2nd image is shown under the flow result). Both ABMOF and ABMOF_LK correctly measure the true flow using a slice duration of only 450 us, equivalent to a frame rate of 22 kHz and a 14 pixel displacement between slices. The consecutive APS image frames were collected at the maximum frame rate of 50 Hz, but because the motion is so fast, even the short DAVIS global shutter exposure of 0.7 ms resulted in visible image blur of several pixels. And since the consecutive frames are separated by 20ms, the images are completely uncorrelated and the resulting flow is meaningless as seen in Fig. 3.10(c).

3.3.3 *Type II experiment result*

The final experiments are from natural scenes that contain a range of directions and speeds. Since we lack the ground truth OF for these natural scenes, we only show the qualitative comparison of ABMOF and ABMOF_LK. These results are shown in Fig 3.11. We use vectors to represent the OF result; color also shows the direction. For clarity, we set $p = 1000$ for ABMOF. The ABMOF and ABMOF_LK produce very similar OF for these natural scenes. For `shapes`, ABMOF flow is quite dense along object edges and the large block size of 21 pixels results in true OF rather than normal flow. For this same scene, ABMOF_LK attaches OF only to object corners; ABMOF_LK misses the OF on the upper right ellipse, but the overall flow is more uniform.



(a) ABMOF (b) ABMOF_LK

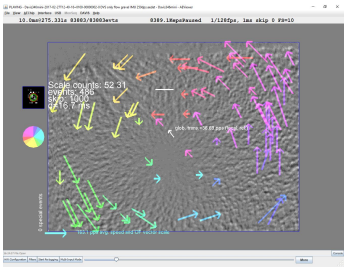


(c) APS image 2 and LK result

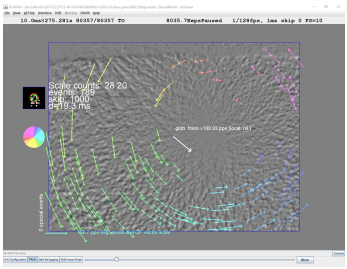
FIGURE 3.10: Result of ABMOF, ABMOF_LK and standard LK on image frames on `pavement_fast`. For 3.10(a) and 3.10(b), we fixed $d = 450$ us, $r = 12$, and $s = 3$, and used standard LK on successive APS images in 3.10(c).

3.4 SUMMARY AND DISCUSSION

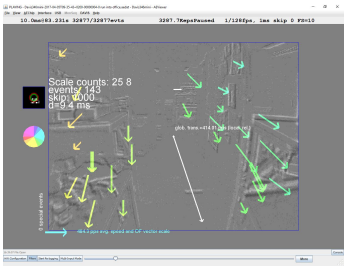
Based on BMOF, we described an improved version ABMOF in this chapter. It is a semi-dense method that computes flow at points where brightness changes. Similar to BMOF, the event representation used in ABMOF is still the event slice. An event slice could be interpreted as a collapsed cuboid. The duration of an event slice is thus defined as the interval between the start timestamp and the end timestamp of this cuboid. ABMOF uses a compressing algorithm called block-matching for OF estimation. “Block" here is actually a set of events that are spatial close on the event



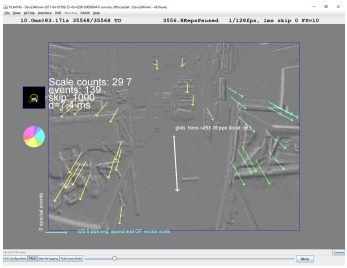
(a) ABMOF for `gravel`



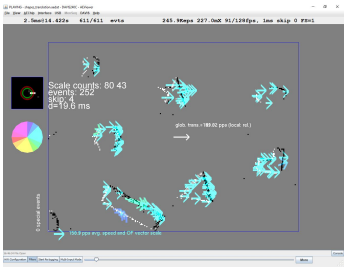
(b) $ABMOF_{LK}$ for `gravel`



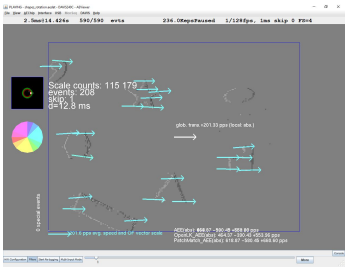
(c) ABMOF for `office`



(d) ABMOF_LK for `office`



(e) ABMOF for `shapes`



(f) ABMOF_LK for `shapes`

FIGURE 3.11: Result of the algorithms on different scenes. All scenes were captured using identical $s = 2$ scales, block size $b = 21$ pixels, using diamond search, with search distance $r = 4$ pixels and using feedback control of *AreaEventNumber* k. OF color and angle represents direction according to the color wheel and vectors' length means OF speed relative to the scale shown at bottom left of each frame. The histogram above each color wheel shows the OF distribution and mean match distance (green circle). The white arrow from center of image shows global average flow. The white statistics text shows the number of OF events for each scale, the number of OF events, the current skip count p, and the last slice duration d.

slice. The method defines two “blocks" and a similarity metric SAD distance to compare them. It manages to find the best block candidate within a finite search area for every incoming event. It is assumed that the appearance of event frames does not change significantly for short times.

Compared with BMOF, we have several improvements. The most important improvement of ABMOF is the adaptive mechanism slice duration of the event slice. This makes ABMOF very robust to the real dynamic scene. The adaptive mechanism consists of two aspects. The first aspect is we proposed a new feed-forward method called *AreaEventNumber*. It adjusts the slice duration based on the local movement rather than the global motion. We compared our generated adaptive slices with the constant time duration slices and it shows that *AreaEventNumber* is more robust than *ConstantDuration*. The second aspect is that the feedback mechanism for slice duration makes the average displacement between two slices close to the half of the search distance. Another improvement is that we use SAD to replace HD in the BMOF design. The last improvement is that multi-scale rather than single-scale is used in ABMOF. Using multi-scale bitmaps allows a larger range of movement speeds to be economically computed and makes the operation more robust to noisy sensor data. All improvements on the basic BMOF achieve a good trade-off between good quality of optical flow estimation and a low computation cost.

The reason we implemented the ABMOF_LK and show the results here is to show the event slices generated by our adaptive method are robust to different scenes and can provide better grayscale images for frame-based algorithms to process. Although ABMOF_LK clearly produces more accurate OF, it is a frame-based OF method that must process every pixel in each frame and produces very sparse output. The gradient-based LK algorithm is also much more difficult and expensive to implement in logic circuits compared with ABMOF.

The result shows that the accuracy of the algorithm mainly depends on the quality of the generated slices. By using the rather large block dimension $r = 21$, ABMOF avoids most aperture problems except on extended edges longer than the block dimension.

The dynamic range of speeds allowed by ABMOF is determined by the search distance r, the number of scales s and the range of slice duration d. The minimum speed could be detected is determined by the maximum d which is 100ms. The maximum speed could be detected is determined by the minimum d which is 1ms and the maximum search range. In any one moment, the range of match distances spans from 0 to $r2^s$ pixels along

each axis, e.g., with search distance $r = 4$ and $s = 3$ scales the OF can span 0 to ± 32 pixels, although the speed resolution decreases as the scale increases. Therefore, the dynamic range of speed is from 10 pixels/s to 32k pixels/s. In our experience, this is a sufficient range to cover real scenes where the camera is rotating, or translating through a cluttered environment with nearby and far objects. With the adaptive slice duration, fast motion can result in slice durations that are fractions of a millisecond, as in the `pavement_fast` example of Sec 3.3.2.2, allowing measurement of speeds > 10k pps. This result was previously only the domain of high-end gaming mouse sensors such as [149]; these are capable of up to several thousand FPS but require active illumination and have less than $50x50$ pixel arrays that are more than a 100 times fewer pixels than the DAVIS used here; also, they only measure global translational flow.

By extrapolating the FPGA hardware implementation costs from [114], we estimate that ABMOF can be implemented on a medium sized FPGA fabric. The resulting IP block could later be integrated together with the sensor in a custom digital core.

The most widely used applications of OF are in optical mouse and video compression, where probably at least a billion ICs have been produced that estimate motion based on block matching. In robotics, most VOD pipelines do not currently use OF, but an economical implementation could enable direct OF based on DVS in hardware, rather than the impressive but expensive software solutions [39, 42, 150]. Recent success in combining DVS with CNNs by using constant event number frames [151–154] also can benefit from the smarter ABMOF slice methods, and the OF could provide useful input channel information to better enable dynamic scene analysis.

Although there is an event skipping mechanism presented in ABMOF, it is a random selection. In the next chapter, we will present the hardware implementation of an event-based corner detector to perform a selection of more informative events.

4

FIRST HARDWARE IMPLEMENTATION OF AN EVENT-DRIVEN CORNER DETECTOR[5]

> *"When someone says 'I want a programming language in which I need only say what I wish done,' give him a lollipop."*
> — Alan Perlis, 1982, "Epigrams on Programming"

4.1 INTRODUCTION

In chapter 3, we introduced a novel block-matching-based optical flow algorithm for event-based cameras. It works quite well with some high-speed motion. It is very robust and adaptive to dynamic scenes. Nevertheless, there are remaining problems that need to be addressed. ABMOF computes optical flow for every event. It has two disadvantages. First, when the event occurs at edges, only the normal component of optical flow can be measured; a local measurement of OF based on gradient can only measure the normal component of flow at edges; the optical flow component along the edge is ambiguous, leading to the *aperture problem*. Second, some events are noise events. Processing them wastes computation power and increases the latency. Therefore, it is unnecessary to process every event. Moreover, it is impossible to process every event in some extreme cases, such as for some superfast motions, such as panning the image across densely textured scenes, when the rate goes above 10MHz. Although a sparsity checking and event-skipping mechanism could filter out some of them, it is not enough. We should have a more informative selection of relevant events on which to compute flow.

In the frame-based vision community, researchers use keypoints (essentially corners) to mitigate the *aperture problem*. Based on the above facts, we thought we might only detect some key events on par with keypoints to compute subsequent information around these events. Therefore, we implemented in hardware a simple but efficient event-based corner detector. This corner detector is designed as a general preprocessing real-time

5 Part of the content of this chapter is from CVPRW 2019 demo [155] and our under-review TCSVT paper on EDFLOW. Copyright © **2019 IEEE**.

hardware logic module IP[1] for event-based algorithms, further to reduce the latency and CPU load on the host processor. We chose EFAST [18] as the algorithm and implemented it on MiniZed FPGA using Vivado SDSoC. The power consumption of the whole system is less than 4W, and the hardware EFAST consumes about 0.9W. Our hardware EFAST implementation processes at most 10M events per second and achieves a power-speed improvement factor product of more than 30X compared with CPU implementation of EFAST. This embedded component could be suitable for integration to applications such as drones and autonomous cars that produce high event rates.

4.1.1 *Corner detectors*

In the Computer Vision (CV) community, corners are often referred to as distinct points in the scene. Corner or keypoint detection is a basic but important topic in the CV community, and it is widely used as a preprocessing step for many CV problems. It extracts the most important information from the image data to avoid processing all pixels and thus decreases the latency and quality of subsequent processing. It underlies many OF, VOD, and SLAM methods. In the conventional CV community, it is well studied, and many popular methods are still used nowadays, such as Harris, FAST, and Scale Invariant Feature Transform (**SIFT**).

Event-based corner detectors. For event cameras like DVS, similar approaches also have been proposed in recent years. Here we give a brief review on event-based corner detection algorithm. Clady [156] in 2015 reported the first event-based corner detector. It detected intersections of local planes fitted to the Timestamp Image (**TI**). Vasco [17] reported event-based Harris in 2016. It adapted the frame-based Sobel-filter Harris detector to binary event patches. Muggler [18] in 2017 reported EFAST. It detects corners on the TI with frame-based FAST. In 2019, Mandersheid [20] proposed a method called Speed Invariant Learned Corners (**SILC**) using a new TI called Speed Invariant Time Surface (**SITS**), which achieves a high contrast around the edge regardless of the scene speed. A random forest classifier detects corners. By using SILC, they decreased the reprojection error by 2X compared to using the normal TI. In 2021, Li [157] posted a Smallest Univalue Segment Assimilating Nucleus) (**SUSAN**)-based corner detector. It also provides a useful summary of software denoising methods.

1 IP is used in the logic design community for “intellectual property" block.

However, the above methods were software developments and used transcendental floating-point serial CPU computation, making them difficult to realize in efficient FPGA logic circuits. For hardware event-based corner detectors, to our knowledge, there is no prior work. As a preprocessing step, corner detection should be done quickly since it is applied to all events. We choose EFAST as our first corner detector hardware implementation because Mueggler *et al*. [18] showed that this algorithm is simple and effective. We found, however, that an even simpler algorithm which we developed and we called SFAST, provides better OF accuracy and its hardware implementation is reported in chapter 6.

The chapter is organized as follows. Sec.4.2 describes the algorithm of EFAST. Sec. 4.3 presents the key points of the methodology of building the accelerator, especially focusing on the transformation strategies. Sec. 4.4 provides the experimental results of the corner detector and Sec. 4.5 concludes the chapter.

4.2 EFAST

FAST is a popular corner detection method in frame-based computer vision. It is widely used because of its good real-time performance. FAST checks if there is a contiguous arc streak whose intensity differences to the center pixel are larger or smaller enough than all other non-streak pixels. If this streak exists, the center pixel is a corner. EFAST originated from FAST. Instead of extracting the corners from the intensity image, EFAST uses the image of the latest timestamps called TI [14] or Surface of Active Events (SAE) in [18]. By using this representation, it is not necessary to accumulate events to mimic a conventional intensity image. Therefore, the data-driven characteristic of the DVS is preserved. Similar to FAST, EFAST also tries to find a corner streak pattern on the TI for every incoming event. A moving corner tends to create a TI of nearby timestamps. Past the corner, the timestamps are older, and hence there is a surface that is like the edge of a cliff with a corner.

To make the algorithm more robust, EFAST uses two circles for checking rather than one circle for normal FAST. To avoid making the image messy, we only show the value of *inner circle* and *outer circle* in Fig. 4.1. The *inner circle* consists of the pixels located on a Bresenham circle with a radius of 3 pixels, and the outer Bresenham circle radius is 4 pixels. Cyan streaks on the *inner circle* and the *outer circle* represent that they have the highest timestamps on the corresponding circles. If we can find these streak pat-

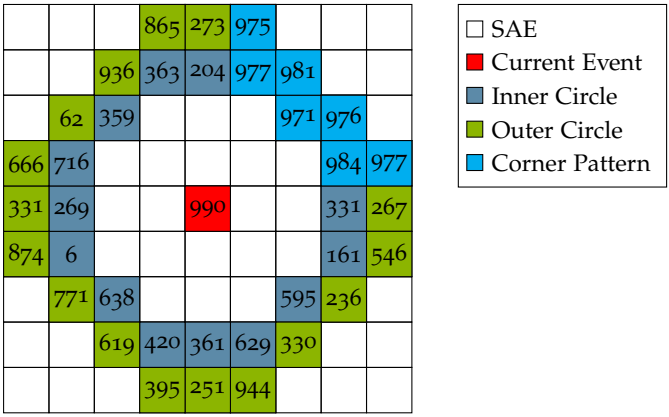


FIGURE 4.1: A simple illustration of the EFAST. In this example, we show a simple situation for a corner event. The cyan streaks on the *inner circle* and the *outer circle* means they have higher timestamps than all the rest pixels of the circle. To make the figure clear, only the time stamps of the *inner circle* and the *outer circle* are shown here.

terns (cyan streaks shown in the figure) on both circles, we call this event a corner event. The length of the streak ranges from 3 to 6 for the *inner circle* and 4 to 8 for the outer circle.

EFAST does not have any complicated mathematical operations such as derivative or floating-point calculation. Only comparators, adders, and some logic gates are required. This makes it suitable for implementation in hardware logic circuits.

4.3 FPGA IMPLEMENTATION

We use MiniZed as our FPGA hardware platform. The MiniZed is a very small and low-cost board. The FPGA on this board is xc7z007sclg225 which mainly targets very light applications. It has the least resources among Xilinx Zynq 7000 family chips. The tool we used to program the FPGA is Vivado SDx. SDx is a tool that is used in SDSoC workflow. SDSoC is more convenient and faster for developing a hardware IP required to work on a Linux system rather than standalone. In this work, our target platform MiniZed needs to access DVS via USB and send the result to a host via WiFi. Therefore, the Linux system is necessary for development, because it provides built-in drivers for the most common USB and WiFi devices. We

thus choose Vivado SDSoC in this work. In this section, we first give an introduction about Vivado SDSoC and Vivado HLS, and then we explore the important strategies to transform a baseline implementation to achieve an efficient hardware design. Finally, we show the memory layout and optimization of this design.

4.3.1 *Introduction of Vivado SDSoC and HLS*

This section introduces the two tools used in this work: Vivado SDSoC first and then Vivado HLS.

4.3.1.1 *Vivado SDSoC*

Vivado SDSoC is a recent tool provided by Vivado to build a more compact design flow for users. The official manual [158] describes SDSoC as follows. "The SDSoC Development Environment is a heterogeneous design environment for implementing embedded systems using the Zynq SoC and MPSoC. It enables the broader community of embedded software developers to leverage the power of hardware and software programmable devices, entirely from a higher level of abstraction. The SDSoC environment provides a greatly simplified embedded C/C++ application programming experience, including an easy-to-use Eclipse IDE and a comprehensive development platform. The SDSoC compiler transforms programs into complete hardware/software systems based on user-specified target platforms and functions within the program to compile into programmable hardware logic. It builds upon customer-proven design tools from Xilinx including Vivado Design Suite, Vivado HLS, Vivado Software Development Kit (SDK), Petalinux, and SDx."

Before the whole system design, a complete SDSoC platform needs to be prepared in advance. It consists of two parts: the hardware platform and the software platform. The hardware platform is generated by Vivado design, and the software platform requires Vivado SDK and Petalinux. Petalinux is only required for Linux support. More details about how to build a customized SDSoC platform for MiniZed is described in Appendix A.1.

Fig 4.2 compare two workflows for designing a customized IP using the normal workflow and SDSoC workflow.

The normal design flow has three stages:

First, use Vivado (Verilog or VHDL) or Vivado HLS (C++) to generate the customized IP and then integrate the hardware IP to a basic hardware

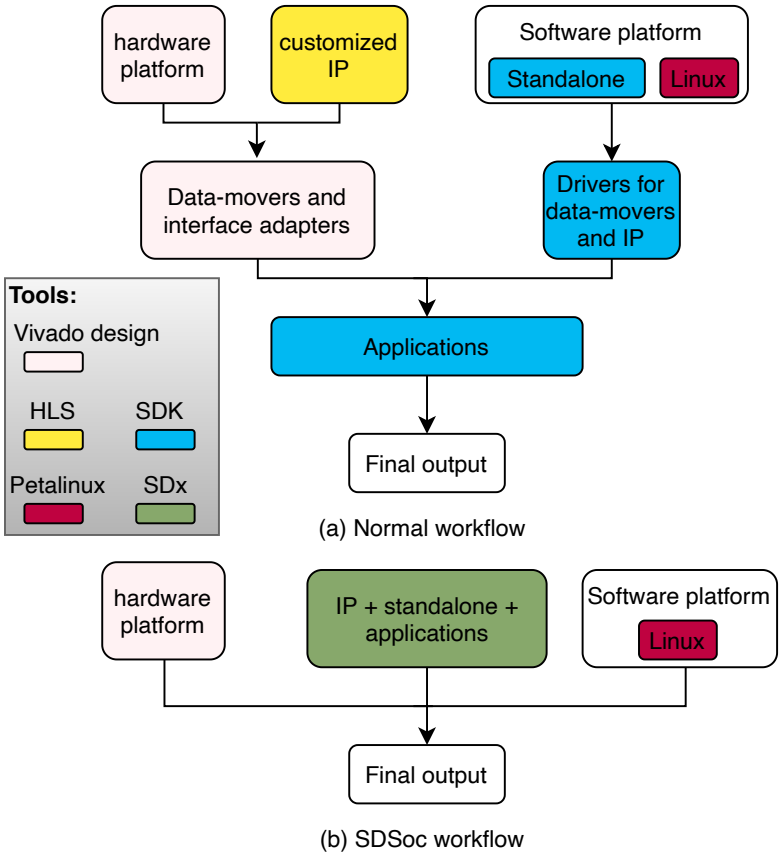


FIGURE 4.2: Comparison between normal workflow and SDSoC workflow. Colors are used to mark which tool is used in the design. Adapted from [158].

design (usually a block design) for a specific hardware platform. This basic hardware design is written as a hardware platform in Figure 4.2(a).

Second, insert data-movers for data exchange between PL and PS manually with Vivado design and write the drive code for data-movers and the customized IP with Vivado SDK. Data-movers are used to exchange data between external memory and IP.

Third and last, develop the ARM software applications to exchange data between hardware IPs and user applications using Vivado SDK.

By Benefiting from Xilinx's latest tool SDx, SDSoC combined all these three stages. As shown in Fig 4.2a, the design of the normal workflow requires three stages and four tools for Linux applications and three tools for standalone applications. The SDSoC workflow, which is shown in Fig 4.2b only requires one stage and three tools for Linux applications and two tools for standalone applications. In the SDSoC workflow, SDx can generate data-movers and interface adapters automatically. Since sometimes the interface provided by the hardware platform is not compatible with the IP, interface adapters are thus required. The drivers for these data-movers and interface adapters are also generated by SDx. SDx provides many primitives, also called *pragmas*, to help advanced users guide the process to generate user-specific data-movers. Interested readers can refer to [159] for more details.

4.3.1.2 *Vivado HLS*

FPGA programming is very complicated. In the early days, FPGA programmers used to implement their design using very low level HDLs. Two most well-known HDLs are Verlilog and VHDL. In recent years, people have made various efforts to develop high-level compilers for FPGA programming. Some of the emerging high-level compilers manage to hide many of the low-level details of hardware design and enable users to describe the hardware design using a high-level language, such as C++. Vivado HLS [160] is an Electronics Design Automation (EDA) tool developed by Xilinx based on this background.

Vivado HLS is a bridge that allows hardware engineers to extend to some algorithm acceleration, and software engineers to accelerate their algorithms on FPGA. However, the programming model of HLS is still substantially different from the programming languages commonly used in software programming, such as C/C++, Java. In software programming, the logic and control operations are written sequentially. The HLS C++ code is more like writing a description of a hardware design, which means that the whole system might not run in the same order that we write the code. The goal of an efficient system is always to make several circuit parts work in parallel. The programmer must specify each unit's input and output variables and the timing schedule between different signals. The different mindsets bring significant difficulties for ordinary users, who are usually more into the software programming mindset, to map their designs into the FPGA's streaming architecture. To generate Verilog/VHDL that

could produce the correct hardware, you must first picture the hardware you want to produce.

Some rules can be helpful to generate an efficient hardware design while writing C/C++ on Vivado HLS. First, although RTL level circuit knowledge is unnecessary, the designer still needs to understand some hardware architectures very well, such as memory partition, memory layout, interfaces, *etc*. Second, designers should always have an overall framework for the whole system. Some critical resources should be controlled. Third, the designers should fully understand how to read the final design report produced by Vivado. The report includes a timing report and a resource report. By reading these reports, the problems/bottlenecks that influence the final performance can be found, and a solution to solve them should be obtained after the report diagnosis. Finally, to achieve the best result, *i.e.*, high performance and low area, designers should always keep in mind the hardware architecture and primary resource units the target provides and use them as a guidance to construct the code.

[161] presents parallel programming on FPGA in detail, and it uses many applications as examples to show the common optimization techniques. [162] provide an overall review of the common tricks used to transform software code to hardware-friendly HLS code. For readers who are interested in this part, we suggest referring them to get more details. In many cases, it is not enough to achieve the desired performance by only depending on HLS directives. Code restructuring is often required. Finding the best code restructuring requires not only understanding the algorithm but also having a sense of the architecture that would be generated by the HLS process [163, 164]. For more details and some optimization tips, see Appendix A.2.

In the first step, we will show the weakness of using the baseline implementation as the HLS code directly. In the second step, we will show how to achieve higher memory utilization by changing the memory layout.

4.3.2 *Baseline implementation*

To obtain an efficient Vivado HLS design, we start from a baseline implementation and explain why this implementation is not hardware friendly. This naive baseline design referred the source code from the GitHub repo[2]. Although the very naive baseline design often cannot satisfy the performance requirement, it is possible for two usages. First, it might be used

2 https://github.com/uzh-rpg/rpg_corner_events

as the entry point for later optimization. Second, it could be used as the testbench for final C and RTL simulation. We show the pseudocode of the baseline implementation in Algorithm 1.

Algorithm 1: EFAST software algorithm (part)

Data: current event e, Surface of Active Events $SAE[2][240][180]$
Result: corner check result $foundStreak$

```
1  x, y, ts, pol ← e;
2  SAE[pol][x][y] ← ts;
3  foundStreak ← false;
4  InnerCircleLoop: for i ← 0 to 16 by 1 do
5      for streak_size ← 3 to 6 by 1 do
6          if streak boundary < neighbors then
7              continue
8          minTS ← minimum of streak;
9          didBreak ← false;
10         if minTS < pixel not on the streak then
11             didBreak ← true;
12             break;
13         if !didBreak then
14             foundStreak ← true;
15             break;
16     return foundStreak;
```

To simplify it, we only show the pseudo-code of the original software implementation to check the inner circle in Algorithm 1. The outer circle checking is similar to the inner circle. From Algorithm 1, we can see that there are too many conditional branches. It has three disadvantages. The first one is that it is not efficient on hardware as it requires hardware resources for all conditional branches even though some of the conditionals are satisfied very few times, making it difficult to share resources. The extra control logic is also required to guide the circuit to switch between different branches. The second disadvantage is that it is not possible to pipeline the module due to the different hardware structures for different iterations. The final problem is that it uses a highly complex process to finish all calculations. This is not good for dataflow optimization and results in hardware resource sharing difficulties since only one instance is im-

plemented on the hardware. Dataflow is a parallel technique in hardware design; similar to pipeline in the operation level, dataflow runs on a higher level, often referring to the function level or module level. By comparing directive dataflow and pipeline, it is clear that it is a function-level pipeline process while the pipeline directive handles resource sharing at the lower level. Check Appendix A.2.2 for more details about dataflow. Therefore, a good architecture of the HLS implementation is that the whole IP consists of several simple processes/functions and is arranged in a producer-consumer order. A process or function is a Processing Elements (**PE**) in the final hardware. Every process might be only responsible for a tiny function. And the pipeline directive can be used inside the PE, and dataflow could be used as a system parallel pipeline to connect all PEs. This is why it is important to make the final system could be guided by the dataflow directive. To address these problems, we redesigned the algorithm from scratch. The rewritten structure of the code is very similar to SFAST, which will be introduced in the next chapter (see Chapter 6).

4.3.3 *Memory layout and optimization*

How to efficiently use the basic 18K Block RAM (**BRAM**) to form an arbitrary size memory is a very interesting question in FPGA implementation. The memory layout of EFAST is 4086x288 bits. These two numbers (4086 and 288) might look strange because the actual size of DAVIS240 is 240x180, and the bit width of the timestamp used in this design is 32 bits. Before explaining the details of why we use this memory layout, some basic BRAM information about Xilinx Zynq FPGA is required. BRAM is an FPGA IP composed of Static RAM (**SRAM**). It is a memory block which is designed to be used in many different configurations. The Zynq series FPGA memory resource is the same as the Virtex7 series. The BRAM in Xilinx 7 series FPGAs stores up to 36 Kbits of data and can be configured as either two independent 18 Kb RAMs or one 36 Kb RAM. Each 36 Kb block RAM can be configured as 64K x 1 (when cascaded with an adjacent 36 Kb block RAM), 32K x 1, 16K x 2, 8K x 4, 4K x 9, 2K x 18, 1K x 36, or 512 x 72 in dual-port mode. Each 18 Kb block RAM can be configured as 16K x 1, 8K x2 , 4K x 4, 2K x 9, 1K x 18, or 512 x 36 in simple dual-port mode [165]. The BRAM can be reshaped using an *array_reshape* directive.

An example of the *array_reshape* is illustrated in Table 4.1. The choice of *array_reshape* is a trade-off. If this value is a power of 2, the address generator consumes less resources since it can be achieved simply by shift

Reshape factor	Memory layout (width*height*depth)	18K Block RAMs	Utilization (%)
1	240*180*4	16	58.59
2	240*90*8	16	58.59
4	240*45*16	16	58.59
5	240*36*20	20	46.88
6	240*30*24	12	78.13
8	30*180*32	16	58.59
9	240*20*36	18	52.08
10	240*18*40	20	46.88
12	240*15*48	12	78.13
15	240*12*60	15	62.50
16	15*180*64	16	58.59
18	240*10*72	18	52.08
20	240*9*80	20	46.88
24	10*180*96	11	85.23
30	240*6*120	14	66.96
48	5*180*192	11	85.23
36	240*5*144	16	58.59
90	240*2*360	10	**93.75**
120	2*180*480	14	66.96
180	240*1*720	20	46.88
240	1*180*960	27	34.72

TABLE 4.1: How *array_reshape* changes the memory layout and utilization of FPGA BRAMs for a memory of size 240x180x4. The column of utilization represents the real occupied percentage of the memory in the whole BRAMs. The red value means that if we set reshape factor 90, it generates a memory layout which only consumes 10 BRAMs and therefore has the highest memory utilization.

operations. Otherwise, a multiplier or even a modulus operator would be inferred. In Table 4.2 we show an example that sets the reshape factor to

Reshape factor	Memory layout (width*height*bitsPerPixel)	18K Block RAMs	Utilization (%)
1	256*256*4	16	58.59/88.89
2	256*128*8	16	58.59/88.89
4	256*64*16	16	58.59/88.89
8	256*32*32	16	58.59/88.89
16	256*16*64	16	58.59/88.89
32	256*8*128	15	**62.50/94.81**
64	256*4*256	15	**62.50/94.81**
128	256*2*512	15	**62.50/94.81**
256	256*1*1024	29	32.32/49.04

TABLE 4.2: How it changes the memory layout and utilization of FPGA BRAMs when we expand a memory of 240x180x4 to 256x256x4. The red value means that if we set reshape factor to 32, 64 or 128, it generates a memory layout which consumes the least BRAMs and therefore has the highest memory utilization. The utilization has two parts, the first value has the same meaning as Table 4.1. The second value means the utilization of the expanded memory.

power values of 2. However, to apply the directive correctly, the width and height of the memory are expanded from 240x180x4 to 256x256x4. If we check the utilization, it consists of two values. The first value calculates the utilization of the original memory, and the second value means the utilization of the expanded memory. It is evident that the expanded memory used fewer logic resources and could adapt to higher resolution applications. However, the highest utilization for the original memory is only 62.5% in Table 4.2 which is far less than 93.75% in Table. 4.1. Therefore, the choice of *array_reshape* is a trade-off between logic resources and memory utilization.

A value that is easy to calculate with subtraction and shift register should be chosen to save resources. For example, we choose 448 here because 448 = 512-64. Thus, if some number x is multiplied by 448, it can be calculated by x*512 - x*64, and it means that we can use subtract and shift registers to obtain the multiplication result. The reason we use 454 here is for memory utilization because the maximum allowed number here is 455.

```
#pragma HLS array_partition partition variable=A cyclic factor=4
```

LISTING 4.1: An *array_partition* example.

An array of HLS is implemented using block RAMs which can at most have two read ports. By partitioning the array, we have independent read ports for each partition, which makes parallelization possible. That is the directive *array_partition* used for. This directive results in implementing several separate memories. As shown in Listing 4.1, this pragma would guide the compiler to generate four separate memories from array A, each of which contains a portion of the array contents. We can think that this optimization provides four times the number of memory accesses each clock, but this is only possible if each memory access can be assigned to precisely one of the memory partitions. For example, a random array access *A[i]* for arbitrary *i* could access data stored in any partition, while an array access *A[4*i+2]* would only access data in the third partition. More complex logic, often called memory banking, can resolve a number of independent accesses *A[i], A[i+1], A[i+6], A[i+7]* and perform these accesses in the same clock cycle [161]. Memory banking requires additional crossbar logic to route these simultaneous accesses in a circuit since *i* can be arbitrary. At compile time, we can guarantee that the constant offsets of these accesses will hit different banks, but the actual banks cannot be determined until it is known. However, more complex logic could implement stalling logic, enabling a set of unrelated accesses to complete in a single clock cycle if they hit different banks. If all accesses happen to hit in the same bank, then the stalling logic can delay the progress of the circuit for several clocks until all accesses have been completed. Lastly, multiport architectures have been designed that can allow a number of guaranteed access completions every clock cycle [166, 167] by replicating data across normal memory on one or two physical ports.

However, using many ports also makes the logic more complicated and thus consumes more logic resources. Similar to *array_reshape*, it is also a trade-off between memory utilization and logic resources.

4.4 EXPERIMENTAL RESULTS

4.4.1 *MiniZed platform*

Fig. 4.3 is the architecture of the whole system. The system consists of two parts: one is the software part that is implemented on the Processing

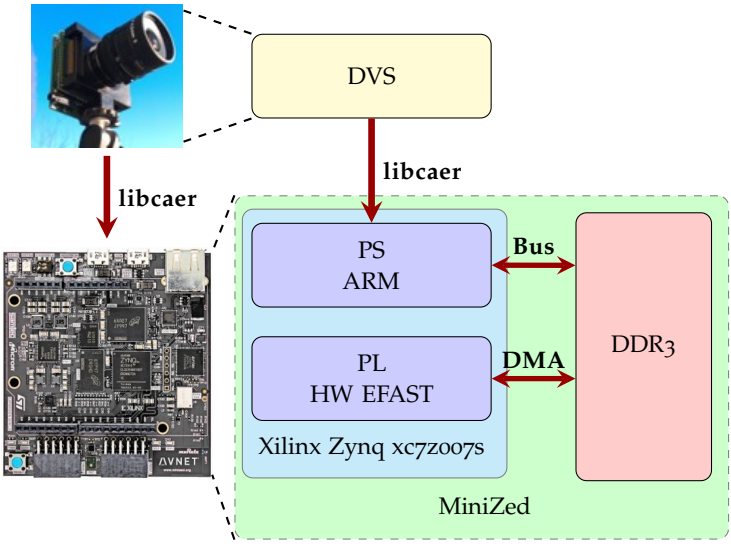


FIGURE 4.3: System Architecture

System (**PS**) with its ARM Cortex A9, and the other is the hardware Programmable Logic (**PL**) part which is a Virtex FPGA. Software on the ARM is mainly used to communicate with the DVS and send the result to the PC. We use *libcaer* to acquire data from the camera, but libcaer depends on libusb, requiring Linux support. A basic and small Linux system is then compiled and run on the ARM. Thanks to Xilinx's Petalinux tool, a Linux kernel for this specific hardware could be compiled efficiently. Data exchange between the PS and PL is based on DDR3. Direct Memory Access (**DMA**) is used to control data access between PL and DDR3. They are implemented as different types of data-movers according to the interface or set by the directive manually, including AXIFIFO, AXIDMA_SG, and AXIDMA_SIMPLE [158].

Detected corners cannot be rendered on MiniZed because no interface on the MiniZed can connect to the external display. We send the result to a laptop by WiFi connection using a dedicated router for the laptop and MiniZed to reduce the transfer latency. The compiled Linux kernel also has built-in driver support for the WiFi chip on the MiniZed board. The laptop hosts a UDP server for the display of the corner detector and DVS data. The laptop is only used for rendering results and providing power.

4.4.2 *Server setup*

The setup of the experiment is shown in Fig 4.4. It consists of four parts: a DAVIS240C, a MiniZed, a router, and a host PC. DAVIS's event packet data is sent to the ARM and stored in the onboard DDR3. The PL reads the data from DDR3 and perform the corner detection and sends the result back to the DDR3 memory. Finally, the corner event stream is sent to the host PC for rendering by the PS via WiFi in a local area network provided by the router.

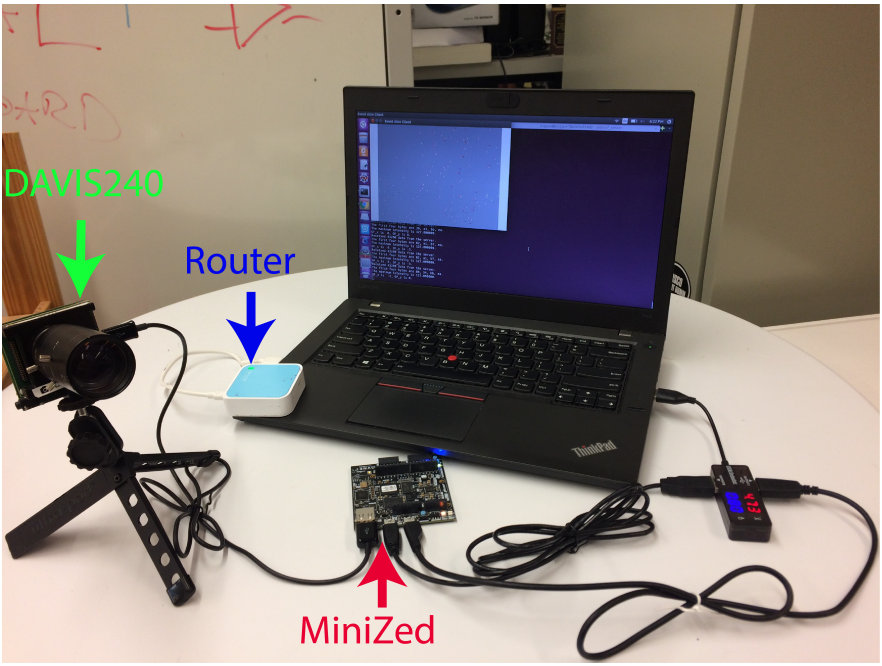


FIGURE 4.4: Setup of the experiment

4.4.3 *Quantitative result*

We use the baseline C++ software version as the testbench for RTL simulation. Vivado HLS generates the test vectors based on the input and con-

verts them to the Verilog test file automatically. After the RTL simulation is validated as correct, we deploy it to the MiniZed.

To validate our method's accuracy, we compare it with the software version. We used the original EFAST GitHub [3] software code as the reference implementation. Both the software version and the hardware version ran on the MiniZed. These two methods use the DAVIS240's live event stream as the input and generate their own output corner event streams separately. Then we compared the hardware results with the software result. We found that our hardware implementation has the identical result to the software implementation. Thus the hardware EFAST has the same accuracy as the software EFAST.

We measured the processing time and power consumption on the MiniZed and compared it with EFAST running on a Ubuntu 16.04 PC with a quad-core 2.3GHz CPU (i5-8259U). EFAST on the PC used the implementation of [18], which runs inside Robot Operating System (**ROS**). It can output the average processing time per event after every packet is processed.

For the power consumption measurement, we use a power meter for MiniZed and the software tool *powertop*[4] for the laptop. We first measured the static power without running EFAST on both of them. After starting the algorithm, the differential value of the power meter and powertop is the power consumption resulting from EFAST.

Resources	**BRAM_18K**	**FF**	**LUT**
Total	57	3038	6046
Avaiable	100	28800	14400
Utilization	**57%**	10%	43%

TABLE 4.3: Resources comsumption of all the modules

The resource utilization percentage of every module and the total number of resources consumed are shown in Table 4.3. It is seen that most resources in this IP are used by memory. This result is still based on the TI with only one polarity of event. The bit width number of the timestamp value here used is 32 bits. If this IP is integrated with other IPs that also

3 https://github.com/uzh-rpg/rpg_corner_events
4 `https://github.com/fenrus75/powertop`

require block memory on the MiniZed, the possible solution would be to decrease the bit width number of the timestamp value.

Platform	FPGA	PC	Comparison
Processing time/us	0.1-0.2	0.6-30	>**6x** faster
EFAST power/W	0.9	5	**5x** more efficient
Total power/W	3.7	17.5	**5x** more efficient

TABLE 4.4: Comparison between EFAST on FPGA and PC

Results are shown in Table 4.4. There are three metrics: processing time, EFAST power, and total system power. *EFAST power* only takes into account the EFAST. *Total power* counts all the parts (DAVIS and MiniZed, or laptop). EFAST running on this small platform is more than 6× faster than running on PC and requires 5× less power consumption. Therefore, the FPGA implementation achieves a power × speed improvement product that is more than 30× higher in FPGA than PC. With a 100MHz clock, the processing time for each event is either 100ns or 200ns (depending on the local context), which means that this system handles up to 10M events per second which are sufficient for most real applications. The EFAST implementation utilized 43% Lookup tables (LUTs), 10% Flip-Flops (FFs) and 57% Block RAMs on the MiniZed.

4.4.4 *EFAST performance in dark environments*

To test EFAST's performance in a very dark environment, we did an experiment where an iPhone smartphone screen light was used as the only light source in a darkened room at night. The smartphone was displaying its normal home screen. The blinds were closed on the window and the computer monitor was shut off during the data capture. The average distance of the scene from the smartphone screen averaged about 3m. The DVS used a lens with an aperture ratio of f/1.3. A light meter was not available, but I estimate, based on my inability to read printed text under these conditions, that the scene illumination was under 1 lux.

The qualitative result is shown in Fig. 4.5. It is obvious to see that even though the event slice generated in a dark environment is very sparse and

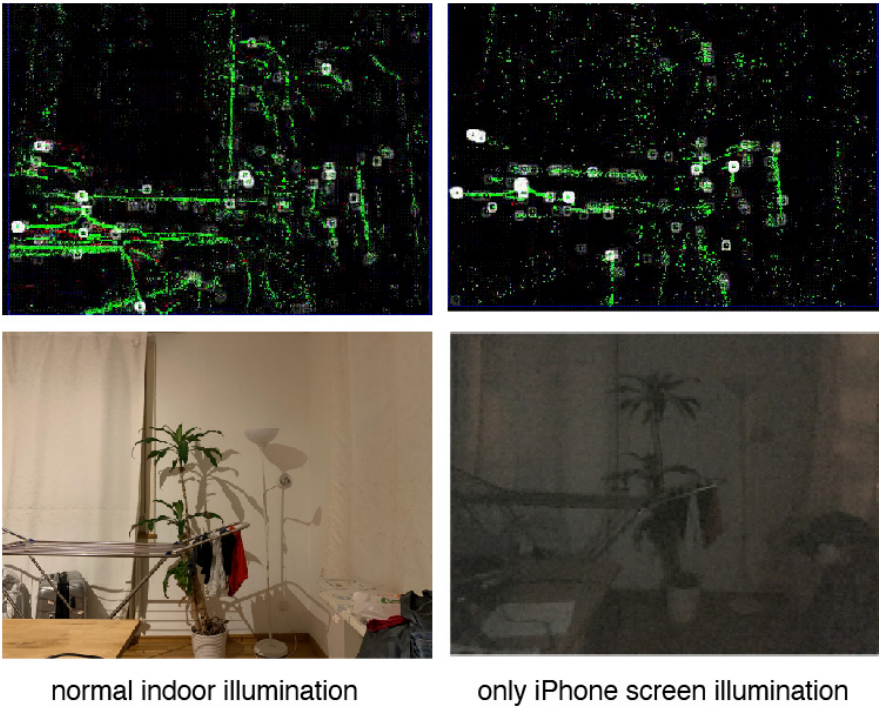


FIGURE 4.5: Two light conditions are used to test EFAST performance. One is the normal indoor condition in the night as shown in the left bottom, the other is the light condition which is only shined by an iPhone XR screen as shown in right bottom. The top row from left to right are the EFAST results. White points overlay on the event slices are the detected corners.

not as clear as the normal condition, EFAST can still detect valid corners and thus proves its robustness in the low light environment.

4.5 SUMMARY

In this chapter, we presented a real-time EFAST corner detector for an event-based camera. We used SDSoC as our development platform and provided a detailed description of this platform and the design flow. Additionally, we also presented the disadvantages of the software baseline

implementation and how to achieve the highest memory utilization using an HLS code transformation. The EFAST IP could process up to 10MHz event rate. It could serve as a useful preprocessing module for many CV problems requiring real-time performance such as OF, VOD and SLAM. We released this IP for free noncommercial use (see Appendix A.7).

We implemented EFAST on the MiniZed. As we discussed in Sec. 4.4.3, if we want to integrate more IPs, the BRAM resources on the board would be not enough. The next chapter will introduce a customized powerful hardware platform `DAVIS346Zynq`.

5

HARDWARE CAMERA PLATFORM DAVIS346ZYNQ

"Computers are useless. They only give you answers."

— Pablo Picasso, 1968

5.1 INTRODUCTION

In Chapter 2, we presented a hardware implementation of BMOF on an Opal Kelly XEM6310MT development board. In Chapter 4, we used a MiniZed FPGA as the hardware platform. The hardware BMOF does not connect to a DVS/DAVIS sensor directly; its data is from a sequencer board which receives jAER files as input. The MiniZed is connected to a DAVIS240C via USB and has limited resources. Constrained by their limited resources, both of these platforms are not suitable for implementing a more complicated optical flow estimator and a corner detector and even more IPs in the future. It is thus important to select a new hardware platform which we call the DAVIS346Zynq camera platform.

So far, existing commercial event cameras such as DAVIS240C or DAVIS346 provide practically no on-board computing (the USB microcontroller actually has no access to the data, which flows directly through the USB IC's endpoint FIFOs). They have small CPLDs or FPGAs on their boards. These chips usually have limited resources. Therefore, they can only handle the AER protocol and event timestamp generation [168, 169]. Data processing is limited to simple denoising.

One possible solution is using external computing resources. These embedded event-based systems consist of a DVS/DAVIS camera and a powerful embedded computer board (most of the MCUs on the board are ARM) such as Raspberry Pi or XODROID, etc. The camera is connected to the computing platform via USB. However, this approach has limitations: First, USB transmission introduces latency and more power consumption, typically at least 1/8 ms latency and power consumption of about 500 mW. Second, many reports show that power consumption of data movement is the highest in the whole system. Third, USB has a bandwidth bottleneck.

The other solution is to increase the onboard computing resource or the near-sensor computing capability. The concept of near-sensor processing

means that raw data is not transmitted outside on USB directly. Therefore, it does not suffer from the USB bandwidth bottleneck. The following two aspects also make near-sensor computing increasingly urgent. On one hand, DVS/DAVIS's resolution has increased from 128x128 to the latest 1280x960 [170], which require more bandwidth for raw data transferring. On the other hand, event cameras are widely used in high-speed applications such as robotics. Therefore, high throughput and low latency while compact and low power consumption methods or algorithms are required.

By placing a powerful computing unit along with the sensor, the raw sensor data could be processed immediately, reducing the latency and power consumption further. Processed output data such as corner locations or OF results are far smaller than the raw data, which decreases the bandwidth requirement. Thus, we design a more powerful camera than the state-of-the-art. It is based on Xilinx SoC Zynq 7100. Since the sensor used in this design is DAVIS346, the camera is called `DAVIS346Zynq`. Sensors send the data to FPGA via AER, and FPGA returns the result data to the outside for rendering or further processing. Raw data is sent to be processed onboard, such as the hardware IPs on our platform. Processed data that includes more specific information such as corners or OF results are then sent out via USB.

`DAVIS346Zynq` targets not only the traditional CV processing but also the deep learning hardware accelerator. The algorithms we introduced in the previous chapters, including ABMOF, EFAST, SFAST which will be introduced in Chapter 6, would use this platform as the test platform.

This chapter is organized as follows: Sec. 5.2 reviews event-based computing platforms. Sec. 5.3 introduces the details of this `DAVIS346Zynq` platform and Sec. 5.4 concludes this chapter.

5.2 PRIOR DEVELOPMENT BOARDS FOR EVENT-BASED CAMERAS

Robotic and Technology of Computers Lab at the University of Seville designed the first development platform for event-based cameras in [171]. In [171], they designed three AER tools mainly for debugging, including a sequencer (PCI to AER), a mapper, and a monitor. The sequencer is used to generate events from synthetic data by PC. The mapper is used for event stream remapping, and the monitor is used to display the event stream on some external device such as a VGA screen. At first, they developed a board based on the PCI interface. The PCI-AER is not very convenient as a portable device, so they develop a new board based on USB and call it

USB-AER board. PCI-AER board only supports sequencing and monitoring, while USB-AER could support all these three modes. The USB-AER board is featured with a Spartan-II 200 Xilinx FPGA and a 512K*32 12ns SRAM memory. The board uses a Silicon Laboratories C8051F320 microcontroller to implement the USB and the MMC/SD interface. A simple VGA monitor interface is also provided to allow the board to act as a monitor (frame grabber) [171]. By configuring the FPGA with different logic programming files, it could be set to several modes [171]:

- Mapper: 1 event to 1 event and 1 event to several events.
- Monitor (frame-grabber): using either USB or VGA as output. For the VGA output, there are two possibilities: B/W VGA, using the VGA connector of the board. And Gray VGA, using a VGA-DAC board connected to the out-AER connector of the board.
- Sequencer: generate live event streams from software.
- Datalogger: allows to capture sequences of up to 512K events with timestamps and send them to the PC offline through the USB bus.
- Player: to play up to 512K events with their timestamps.

The main disadvantage of this board is that the RAM is too small (only 2MB), which can only be used for a short recording. Besides PCI-AER and USB-AER boards, they also developed an AER-Switch board. AER-Switch board supports 1 to 4 or 4 to 1 AER interface conversion.

In a master project, Raphael Berner designed a useful USBAERmini2 board whose IP served as the basis for all the subsequent logic used in DAVIS cameras [138]. The mini USB-AER board is a simplified version of the standard USB-AER board. It does not have FPGA and MMC/SD on board. This means that the board's function is in the software implementation, making it slower than the standard USB-AER board. This simple and low-cost board was used in many INI projects subsequently as a handy way to monitor and sequence events.

A comparison among the above development platforms for the event-based camera is presented in Table 5.1.

In 2015, [173] designed a new FPGA-based platform for event processing. This hardware platform featured a Lattice ECP3 FPGA with 17K logic gates, an ADC, an IMU and a Cypress FX3 USB 3.0 Super-Speed microcontroller. They also provided the hardware implementation of two low-level algorithms on this platform. The two algorithms consist of a time-spatial

Features	Debugging USB AER Tools		
	USBAERmini2 [138]	USB-AER [171]	AER Opal Kelly board [172]
Capacity	N/A	2M bytes	128M bytes
Event width	16 bits	16 bits	16/32 bits
Monitor	5Mevps (Peak 6Mevps)	N/A	6.5Mevps (Peak 8Meps)
Player	N/A	3.75Mevps	6.5Mevps
Logger	N/A	12Mevps	20Mevps
Power (mA)	60mA	30mA	32mA

TABLE 5.1: Comparison between different development boards for event-based camera.

correlation denoise algorithm and a simple tracking algorithm. And they use a USB3.0 interface for event transfer to the laptop/PC.

In 2016, [172] presented a new AER Opal Kelly board that integrated sequencing, logging, monitoring, playback, and merging, forming an all-in-one package. Additionally, the onboard memory was increased. In 2017, [174] reviewed the computing platforms for neuromorphic vision sensors, and provided a comprehensive survey of neuromorphic computing platforms.

In 2019, [175] gave a detailed report on the FPGA implementation of several event-based algorithms that consist of Background Activity for denoising, Mask Filter for hot pixel removing, Rectangle cluster for tracking, and Object Motion Detection on two FPGA platforms. These two platforms are Xilinx FPGA-based AER-Node [176] and Lattice FPGA-based DevBoardUSB3. And they showed that the hardware implementation is up to 570% faster while the power consumption is 5x more efficient than the software implementations in jAER running a 2.2GHz i7 CPU. All hardware interfaces of these algorithms on AER-Node are compatible with the AER protocol, but DevBoardUSB3 uses a word-serial protocol.

All previous boards/platforms suffer from either small memory storage [171] or a low/middle-grade computing unit [172, 173, 175, 176], this work `DAVIS346Zynq` uses 512MB Double Data Rate (DDR)3, 1GB flash and the highest grade SoC of Xilinx Zynq family.

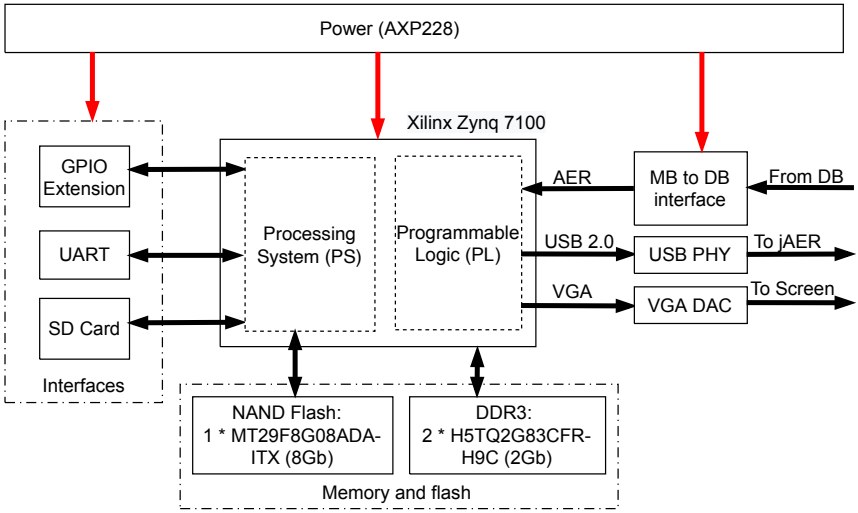


FIGURE 5.1: Hardware block diagram of the mother board on DAVIS346Zynq. It mainly consists of the following parts: power, interfaces, SoC(PS and PL), and memory.

5.3 DAVIS346ZYNQ

My DAVIS346Zynq was manufactured with a 6-layer PCB and assembled at a cost of about $3k for 4 samples, it holds a DAVIS346 sensor chip [177], the most powerful SoC FPGA of the Xilinx Zynq 7000 family (*XC7Z100*[1]), a USB2.0 high-speed PHY (*USB3320C*), 512MB of *H5TQ2G83CFR − H9C* Dynamic RAM (DRAM), an Secure Digital (SD) card slot, the power module, and 1*GB* of *MT29F8G08* NAND flash. This Zynq 7100 SoC has a Kirtex-7 FPGA on the PL and a dual-core 800 MHz ARM Cortex-A9 on the micro-processor PS.

This section introduces the whole system architecture first and then describes the power circuit, DDR3 controller, flash, GPIOs, and customized circuits, including DAVIS controller, VGA, and USB.

1 Costing about $500

5.3.1 *Hardware architecture*

`DAVIS346Zynq` includes two parts: Mother Board (MB) and Daughter Board (DB). The hardware architecture of this camera's MB is shown in Figure 5.2. The DB uses the existing DAVIS346 camera's DB. The MB connects to the DB through MB to DB interface. AER data is sent to the FPGA, and the AER state machine in the FPGA PS processes it and sends it to other IPs for further processing. The PS part is mainly responsible for interface control, memory access, and human-machine interaction.

By providing SD card access, recording the live stream is supported by `DAVIS346Zynq`. Previously, the DAVIS240 or DAVIS346 cannot record data by itself. When using them to record data, another embedded computer such as Raspberry Pi running with cAER or a bulky laptop running with jAER is required. `DAVIS346Zynq` can record data by itself.

It also supports a VGA display. With VGA support, we do not need to connect the camera to the PC first and then use jAER to visualize it while debugging. It can show the generated DVS event video on a monitor very conveniently. This built-in VGA circuit render function makes it more independent.

512MB DDR3 memory on the board makes it possible to run Linux systems on the board. Some system-level functions or packages, such as ROS, are then possible to be deployed.

The customized USB 2.0 IP is entirely written in Verilog and supports the high-speed mode. For regular USB controller ICs such as Cypress FX2/FX3, it integrates an ARM and the USB circuit, so the software is required to respond correctly to the USB host. However, in this design, all functions are implemented in the hardware circuit, and thus no PS intervention is required.

Other features include 1 UART for debugging, several buttons and LEDs as simple human-machine interfaces, and 1GB Nand flash to store some start-up applications and record the live stream online.

The schematics of this design and the FPGA code, including USB and SPI master for DVS configuration, are released as an open-source project; see Appendix A.7.

5.3.2 *Power and storage circuits*

The power circuit. The power circuit is a core part of the system. It functions as the "heart" of the board. This circuit should be very stable. Xilinx

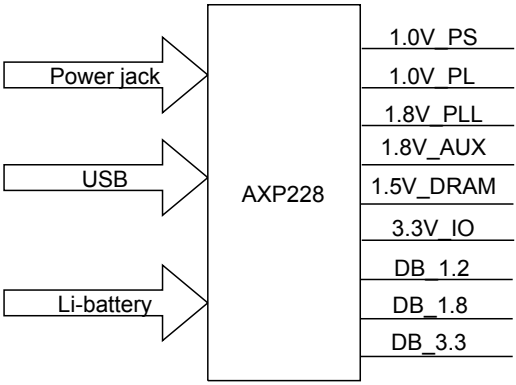


FIGURE 5.2: Power circuit block in DAVIS346Zynq.

Zynq 7000 series requires multi-power supply channels, including 1.0V for PS and PL core circuit, 1.8V digital power for the auxiliary circuit of PS and PL, 1.8V analog power for PLL circuit on the chip, 3.3V for IO, and 1.5V for DRAM controller on SoC and memory chip. The power-on sequence is important. The recommended power-on sequence for the PS part is the core circuit, then auxiliary and PLL circuits together, then the I/O supplies to achieve the minimum current draw and ensure that the I/Os are 3-stated at power on [178]. The recommended power-on sequence for PL is similar to PS. To satisfy the power requirement for the board, we choose $AXP228$ as the power supply chip. $AXP228$ is a highly integrated Power Management IC targeting applications that require multichannel power conversion output designed by XPower company. It accepts three types of power input: external input from a power jack, USB power input, and Li-battery power input. The Li-battery power input gives the system more portability than the regular event-based camera. The accepted input voltages range from -0.3V to 11V. For output, it supports at most 20 channels power outputs, including 6 channel bucks. The maximum output current for these 6 bucks ranges from 1.8A to 6A. All voltages can be dynamically adjusted by programming via I^2C protocol. The power sequence can also be programmed. To ensure the security and stability of the power system, it provides multiple channel 12-bit ADCs for voltage/current/temperature monitor and integrates protection circuits such as over-voltage protection, under-voltage protection, over-temperature protection, and over-current protection. Besides, $AXP228$ features an intelligent power select circuit to transparently

choose the power path between USB and Li-battery. The control and status registers residing on the chip can be easily accessed by I^2C bus. $AXP228$ proves it a very suitable power chip for `DAVIS346Zynq`.

DDR3 controller Xilinx Zynq 7000 SoC supports $DDR2$, $DDR3$, and $DDR3L$ devices and consists of three major blocks: an AXI memory port interface, a core controller with transaction scheduler, and a controller with digital PHY [179]. Thanks to the integrated DDR controller, we do not need to take care of the complicated communication protocol too much. However, there is one thing that is essential when designing the circuit: It is important to ensure the distance from the memory chip to the SoC is as close as possible since the clock on the bus is 666MHz. To improve the anti-interference while running at this high frequency, the physical layer of the bus uses differential lines for transmission. Therefore, the wire lengths for all data signals, clock signals, and control signals should be identical (see Sec A.6). In this work, $DDR3$ is the $2x2\,Gb(512MB)H5TQ2G83CFR$ manufactured by Hynix.

GPIOs PS has limited IO pins, and they are called multipurpose IOs. All IO pins can be used as a general-purpose IO pin and configured as a dedicated function pin. The dedicated functions include UART, SPI, SD Card, NAND flash and Ethernet, etc. But specific care should be taken when using an SD Card as the boot device. In this case, the SD memory card must be connected to SDIO areas belonging to pins 40-47. For more details about IO and boot device configuration, we refer the readers to [179]. The IO configuration used for this design includes 1 SD Card, 1 NAND flash, 1 USB, and 2 UARTs.

Flash memory Flash memory is a critical device for the system because the SOC does not provide large on-chip non-volatile storage. All configuration and applications or operation systems must be stored in flash. In this work, we use the NAND flash as our boot-up device. Some rules should be obeyed if using it as the boot-up device.

The first rule is to tie the bootstrapping pins in the correct state. These pins affect boot configuration options. Immediately after the reset is finished, the SoC samples the bootstrap pins and optionally enables the PS clock PLLs. Then, the PS begins executing the BootROM code in the on-chip ROM to boot the system [179]. After executing BootRom code, the hardware will handle the system control to the flash device. Which flash device would be selected is according to the strapping pin hardware sampling. Therefore, it is essential to tie the strap pins to the correct value to make the flash chosen as the boot device.

The second rule is to choose a compatible flash. A flash could be read and written by the SoC during normal operation, but it might not be compatible with the boot operation. It is important to check if the flash used is officially supported. Supported flash device list can be checked on Xilinx website[2].

Two common types of flashes are QSPI flash and NAND flash. QSPI flash has fewer wires and thus more convenient, but the memory density is often low. NAND flash has higher memory density but has more pins and complicated protocol. For `DAVIS346Zynq`, we selected NAND flash as our boot device and chose $MT29F8G08A$ manufactured by Micron. This flash is an $8Gb$ $(1GB)$ flash, and the data bit width is 8bit.

5.3.3 *DAVIS controller*

5.3.3.1 *AER protocol*

AER [169, 180] is an asynchronous communication protocol used to transfer data between two bio-inspired systems/devices/chips. It was first proposed by Sivilotti [180] in his Ph.D. thesis in 1991. AER is inspired by the transmission of neural information in a biological neural system by an electrical spike. Currently, it is widely used in the neuromorphic engineering community for the spiking sensors and SNNs, such as DVS, DAS [181] and Dynapse [182, 183].

Two sets of signals are used in this protocol: data signals, and the asynchronous handshaking signals. Data signals are 11 bits for $DAVIS346$ sensor chip. All current iniVation DAVIS sensors employ a word-serial data format, meaning that the x(column) and y (row) addresses can not output concurrently, but separately one after the other. The Most Significant Bit (MSB) of the data is used to distinguish between row address and column address. The sensors employ a row-wise readout scheme, so a y address will always be followed by a series of one or more x addresses. The column address also contains the polarity information bit. The handshaking consists of two signals: REQ and ACK.

5.3.3.2 *Hardware architecture*

The hardware architecture of the DAVIS controller is shown in Figure 5.3. It started from the logic developed in EU SeeBetter and Visualize projects [169, 184]. It consists of several state machines, including DVS

2 https://www.xilinx.com/support/answers/50991.html

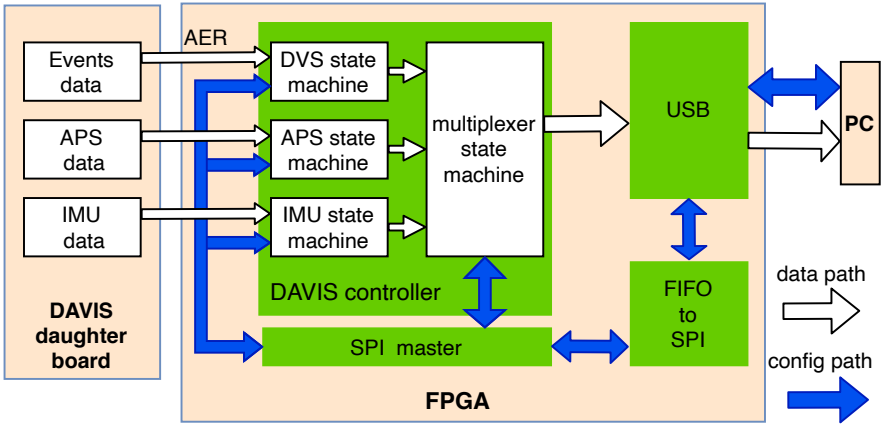


FIGURE 5.3: The hardware architecture of the DAVIS controller.

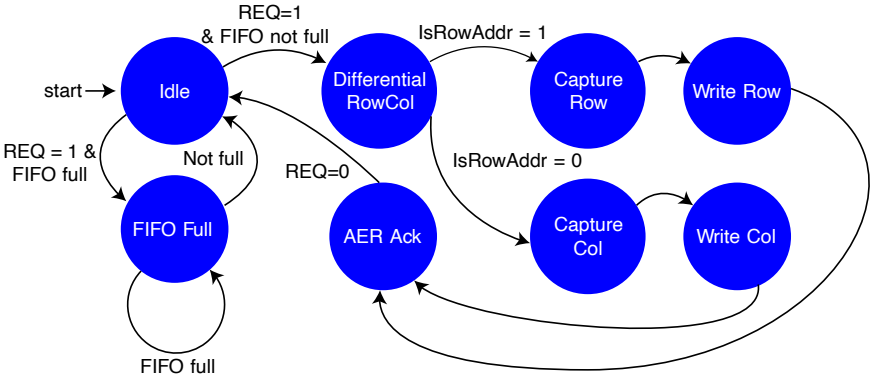


FIGURE 5.4: DVS state machine.

state machine, APS state machine, IMU state machine, and multiplexer state machine. All these state machines could be configured by an SPI master. A PC can control all state machines via the USB interface. The USB is also implemented on FPGA in Verilog. The details of the USB part are shown in Sec. 5.3.5. DVS state machine, APS state machine, and IMU state machine read the data output from the DAVIS DB. The multiplexer state machine selects the data source according to the hardware configuration. It also generates the timestamps, packs the events/APS/IMU and timestamps in a specific format, and sends them to the USB controller.

The state transition of the DVS state machine is presented in Figure 5.4 [184]. It is mainly used to respond to the AER protocol correctly. In the *Idle* date, both REQ and ACK are deasserted. When the sender asserts the REQ line, the state machine checks if the event data FIFO is full. If it is full, then this data will be skipped and change to *FIFO Full* state. It can only jump back to the *Idle* state unless the FIFO is not full again. Since the row address and column address share the same bus, there is a state called *Differential RowCol*. It changes to the read row address and column address according to the MSB. The *Capture Row* and*Write Row* states are used to read and store the row address data to the FIFO. Similarly, *Capture Col* and *Write Col* are used to read and store the column address data to the FIFO. *AER Ack* confirms having read the data by asserting the ACK line. It would jump back to *Idle* if there is no more data request.

Maximum event rates. We estimate the maximum event rate according to the DVS state machine and the multiplexer state machine. We consider the two most extreme cases in which events are triggered simultaneously: i) they are on the same row; ii) they are not on the same row. First, they are in the same row. The events would be generated in order as following: $timestamp, y_1, x_1, x_2, x_3..., x_n$. In this case, there are n events generated, and all these three events have the same row address $y1$. The interval between the first two events is 5 cycles. The intervals between all other events are 8 cycles. Second, they are from different rows. The events would be generated in the order: $timestamp, y_1, x_1, y_2, x_2, ..., y_n, x_n$. In this case, there are n events generated, and they are from different rows. The intervals between every two events are 20 cycles. All state machines are running at 100MHz. Therefore, in the first case, the maximum event rate is 20Meps, and in the second case, the maximum event rate is 5Meps.

5.3.4 *VGA for events rendering*

VGA is a video standard proposed by IBM in 1987. It is mainly used for transferring a PC's graphics card video signal to an external monitor. It is widely used in not only CRT devices but also modern LCD monitors. Different from the latest HDMI, VGA uses an analog signal. Early neuromorphic chips directly drove a multisync VGA monitor to display their output, *e.g.*, the retina, motion chip, stereo chip, or audio chip pixel array state [185]. To make the `DAVIS346Zynq` could visualize the event stream more conveniently, we implemented a VGA inteface. The details of the VGA protocol and timing diagram are shown in Appendix A.3.

5.3.4.1 *The schematic of VGA*

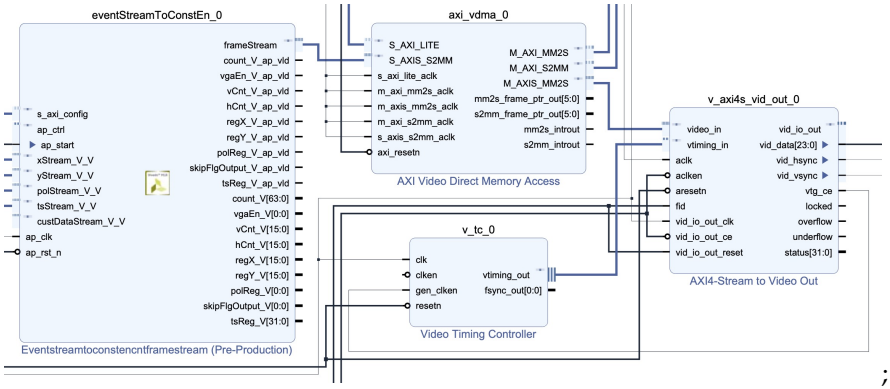


FIGURE 5.5: The VGA schematic in Vivado.

Fig. 5.5 shows the VGA schematic. It consists of four blocks: *EventStreamToConstantCntFrameStream*, *AXI Video Memory Access*, *Video Timing Controller* and *AXI4-Stream to Video Out*.

EventStreamToConstantCntFrameStream is used to convert the event stream to a fake frame stream for rendering. They are accumulated in a constant duration. The default value is 10ms, but it can be changed by register setting. It supports three display modes: event slice mode, TI mode, and customized mode. The event slice mode sets the event count as the pixel value. This mode is the default display mode. The TI mode sets the most recent timestamp as the pixel value. The user could also set the customized data such as optical flow results as the pixel value. Three modes could be switched at any time. This *EventStreamToConstantCntFrameStream* IP is written in Vivado HLS C++ code. The source code is released on github[3].

Xilinx provides the other 3 IPs. *AXI Video Memory Access* is a DMA that copy the frames to the *DDR3*. There is a specific area on *DDR3* used for display memory. It outputs the data in a *AXI4* stream format. *Video Timing Controller* provides the VGA timing as we shown in Fig. A.7 and Table A.1. It supports a variety of resolutions. The configuration window of *Video Timing Controller* is shown in Fig. 5.6. We used 800×600 as our resolution. *AXI4-Stream to Video Out* generates the HS and VS according to the timing. The data is sent to the VGA DAC board for digital-to-analog conversion.

3 https://github.com/wzygzlm/hls_abmof/tree/master/EventStreamToFrameStream

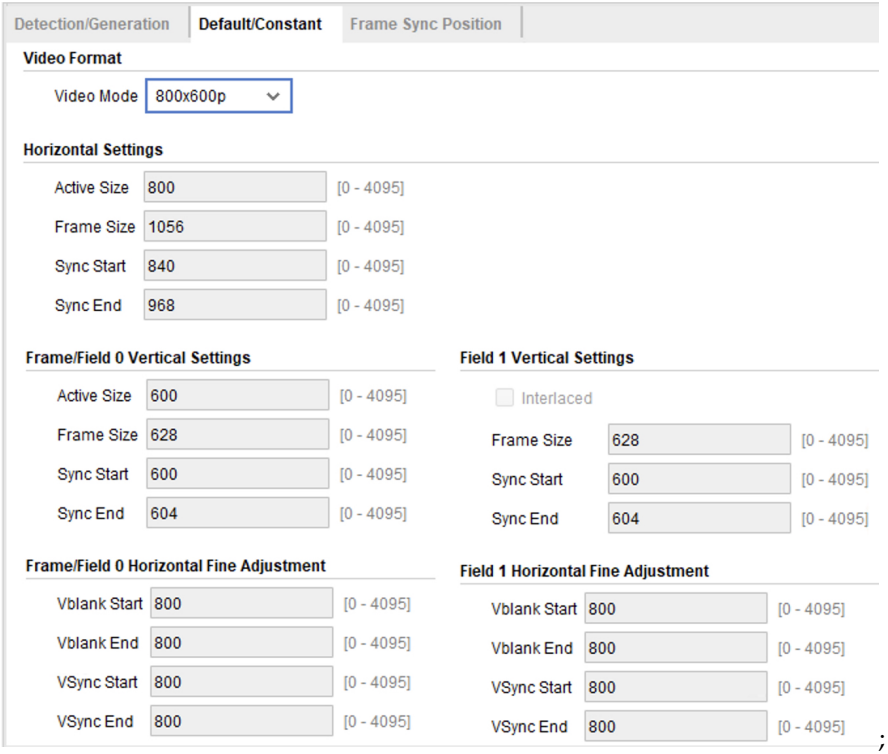


FIGURE 5.6: VGA timing settings for Video Timing Controller IP.

5.3.5 *Reimplementation of the USB controller*

USB is a modern serial bus for communication between PC and its peripherals proposed by Intel, IBM, Microsoft, etc. It is widely used in PCs, phones, and other embedded devices and has almost become a standard interface because of its plug-and-play and few wires connections. Although the PS on the SoC has a built-in USB controller, it does not work on the board due to some mistakes in the schematic. Therefore, in this work, we reimplemented a USB 2.0 controller circuit on FPGA. It supports the USB 2.0 high-speed mode (480Mbit/s). Compared with the existing USB solution (Cypress FX2) on DAVIS240C which uses software to handle the complex enumeration process, `DAVIS346Zynq` responds by the circuit automatically and does not require the software's intervention, which makes it more convenient. This effort took quite a significant amount of time. The

groundwork of our design is an open-source project Ultraembedded[4]. For the details of the USB 2.0 protocol, see Appendix A.4.

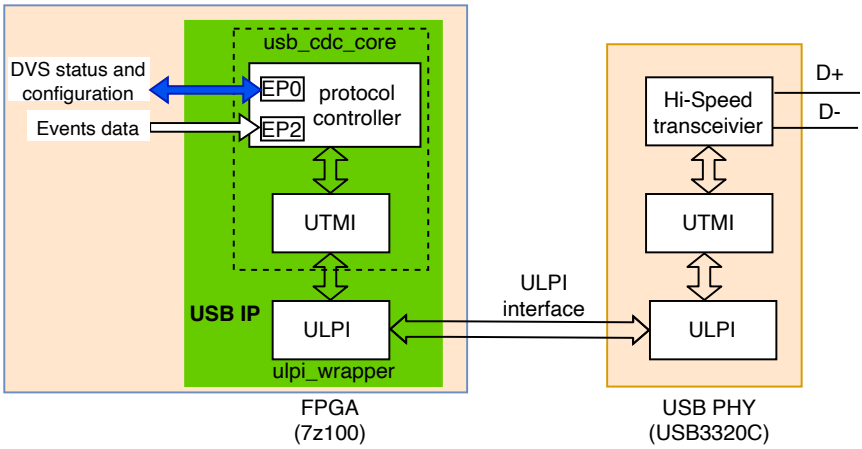


FIGURE 5.7: The hardware architecture of the USB device.

5.3.5.1 *Hardware architecture*

The hardware architecture of the designed USB device is shown in Figure 5.7. It consists of four parts: USB protocol controller, USB Transceiver Macrocell Interface (UTMI) [186], UTMI+Low Pin Interface (ULPI) [187] and USB PHY. The first three parts are digital circuits, but the PHY circuit is a mixed-signal circuit, and thus we implemented the first 3 parts on FPGA, and USB PHY is implemented using a small development board. The small development board is also shown in Figure 5.8. The board used here is *USB*3330 accessory board designed by waveshare[5]. This board acts as the USB high-speed external PHY device for the ULPI interface and features *USB*3300. ULPI is a reduced version of UTMI because UTMI has more than 50 pins which makes it not convenient to communicate with other chips, but ULPI has only 12 pins. That is the reason why a ULPI wrapper is added at the bottom of the USB IP. From Figure 5.7, we can see that the protocol controller and UTMI are implemented together as usb_cdc_core and the ULPI is implemented in another independent IP ulpi_wrapper.

4 https://github.com/ultraembedded/core_usb_cdc
5 https://www.waveshare.com/wiki/USB3300_USB_HS_Board

The protocol controller is responsible for implementing the USB protocol. This work used two endpoints: endpoint 0 (EP0 in Figure 5.7) for sending USB/DVS configuration and reading USB/DVS status; endpoint 2 (EP2 in Figure 5.7) for sending event data to the PC. However, in the original implementation, it only supports standard requests, which means it is easy to send the configuration to the USB device and read the USB's status. However, configuring DVS or obtaining status from DVS is impossible; we thus add the vendor request support for the endpoint 0 to the original design. Since this device only supports the FIFO interface while the DVS configuration supports only the SPI interface, a module called USBFifoToDVSSPI (see Figure 5.3) is used to convert the interfaces between them is designed. Endpoint 2 reads the data generated from the AER logic handler and then packets it according to the USB transfer type. The output packet data is then sent out via the UTMI interface.

Ulpi_wrapper converts the UTMI interface to the ULPI interface. USB PHY interprets the data from the ULPI interface and then encodes them and sends it out. USB PHY is the transceiver chip, and it is responsible for converting the digital signal to Non-Return-to-Zero Inverted Code (NRZI) and then transferring them to the corresponding electrical waveforms.

5.3.6 *Final PCB*

The final assembled board is shown in Fig.5.8. DAVIS controller handles AER handshaking, timestamp generation, and USB interfacing with jAER. The PS runs a bare-metal program that lets us program registers over the UART port, but it is otherwise idle. Details about the UART configuration are described in Appendix A.5. Until now, hardware IPs such as ABMOF and SFAST has been implemented on PL, along with a customized USB2.0 IP and other IPs such as a USBFifoToDVSSPI and a SPI master for DVS configuration. A few cells are designed as logic schematics (VGA) or VHDL/verilog (DAVIS controller, USB controller, USBFifoToDVSSPI, and SPI master). Benefit from its plentiful resources, it is possible to incorporate more hardware IPs in the future. The evaluation result of `DAVIS346Zynq` including resources utilization and power consumption is shown in Chapter 6.

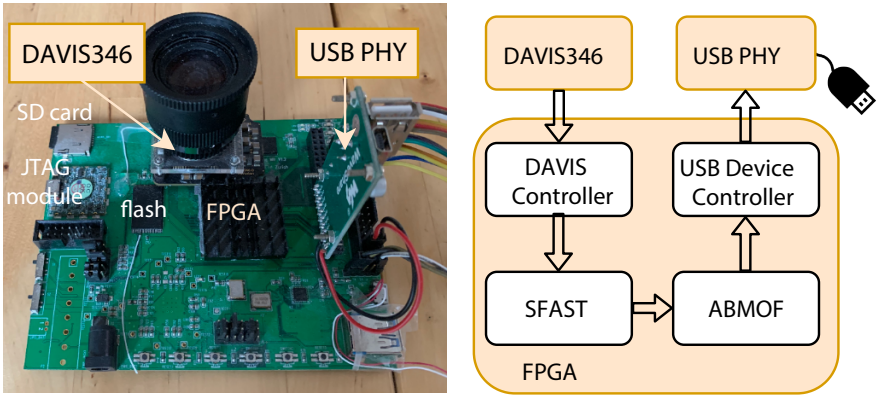


FIGURE 5.8: The prototype `Davis346Zynq` event camera platform. It consists of three parts: DVS as the input, FPGA as the hardware processing and USB as the output. SFAST and ABMOF are the two event-based hardware IPs shown in chapter 6.

5.4 SUMMARY AND DISCUSSION

In this chapter, we show a novel powerful camera called `DAVIS346Zynq`. It features a 346x260 DAVIS sensor and a powerful Xilinx Zynq Soc 7100. For peripherals, it supports UART for configuration, SD Card for storing event data, and NAND flash for system boot and large applications or operation systems. This camera benefits from the USB and VGA interfaces to support output data either via USB or VGA to an external screen display. It supports two play modes: live mode and playback mode (sequencer). Plenty of resources on the PL part of the SoC make it possible to test and verify many algorithms. This board supports the most common interfaces and has a powerful ARM, FPGA, and large memory storage. It is enough to be a general platform for a neuromorphic vision system. For the future, a necessary improvement will be to make the board smaller, lighter, and more mechanically robust so that it can be more suitable for an embedded neuromorphic computing platform for drones and robotics.

The debugging for this board was a time-consuming task. During this process, I met many problems such as SoC could not be recognized by PC via FPGA, reading and writing mistakes for the DDR3 memory test, USB protocol circuit design at the lowest level, and the system could not boot

from NAND flash, *etc*. However, it is a really painful but happy process. However, I indeed obtained a more profound understanding of these circuits during the debug process. More details about the debugging story are shown in Appendix A.6.

The next chapter will show the hardware implementation of two event-based algorithms (an optical flow estimator and a corner detector) on `DAVIS346Zynq`.

6

HARDWARE IMPLEMENTATION OF ADAPTIVE BLOCK MATCHING FLOW AND CORNER DETECTOR ON DAVIS346ZYNQ¶

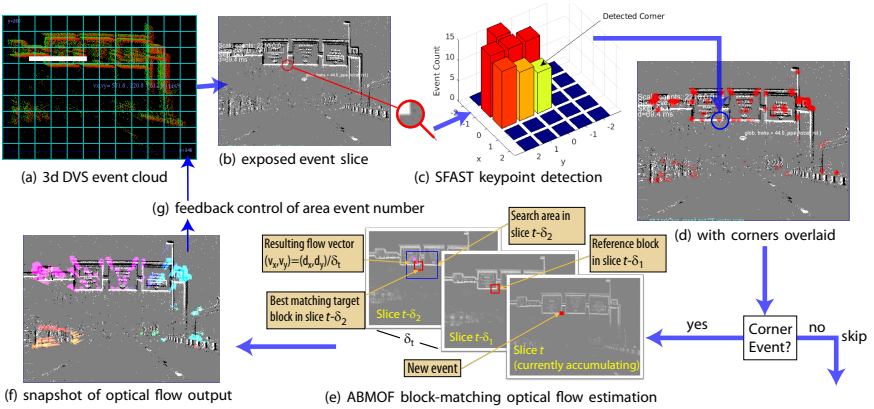


FIGURE 6.1: Overview of the entire EDFLOW camera. (**a**) 50 ms 3D space-time DVS event cloud from a camera mounted on a car dashboard. (**b**) Automatically exposed slice of brightness change events[1] with magnification of one corner. (**c**) The 3D shape on the event count slice for the keypoint. (**d**) The slice with all detected keypoints overlaid. (**e**) Illustration of the basic (single-scale) block-matching optical flow. (**f**) Snapshot of final optical flow vector output computed at the detected keypoints of 50ms slice. (**g**) feedforward and feedback control flow diagram of slice exposure duration.

6.1 INTRODUCTION

In Chapter 4 we have mentioned that computing flow on every event is not efficient and it might also suffer from the *aperture problem*. To compute

¶ The substantial content of this chapter has been submitted to TCSVT.

1 Generated by *run1_test/Davis346B-2016-12-15T11-54-56+0100-00INX006-0 back from airport.aedat* sequence from DDD20 [188] at time 213.7s.

on more informative events, we used EFAST as the corner detector and implemented it on FPGA. The measurement of OF is more robust at the locations of corner-like features called *keypoints*. At keypoints, the OF can be unambiguously determined. The algorithm first detects keypoints and then measures OF only at these reliable keypoints. These same keypoints are used in semidense VOD and SLAM pipelines, and OF can help the matching processing converge faster and more reliably. One significant drawback of EFAST is it operates on TIs while ABMOF operates on event slices. It makes EFAST not compatible with ABMOF. We thus proposed an improved corner detector called SFAST 6.3.2 based on EFAST.

Combining SFAST with ABMOF, we built EDFLOW, which outputs along with the original events a stream of keypoints and the OF of these keypoints on the powerful new camera platform described in Chapter 5. Fig. 6.1 illustrates EDFLOW. It consists of two main parts: a keypoint detector and an OF estimator. Corner events drive OF estimation at informative locations and thus reduce the data rate for OF estimation so that even high event rates such as those encountered during high speed mobile robotics can be processed in real time without skipping informative events for OF. It has two advantages. The first is that by filtering out a lot of noncorner events, it saves bandwidth for later OF computation. The second is that it helps mitigate aperture problems.

EDFLOW uses SFAST for corner detection and ABMOF for OF estimation. Instead of calculating optical flow on all 3 scales like ABMOF, SFAST only detects keypoints on the coarse scale, which is important for reducing the cost of the implementation.

In Chapter 2, we presented a basic DVS BMOF FPGA implementation that does not support adaptive duration event slice or multiscale BM, and described the improved ABMOF software algorithm in Chapter 3. This chapter greatly extends the hardware implementation in Chapter 2 to support all the advanced ABMOF features for a sensor with with 5X more pixels. Additionally, we combine the hardware ABMOF with an SFAST corner detector for better accuracy than with EFAST corner detection.

The rest of this chapter is organized as follows. Sec. 6.2 reviews the prior hardware OF implementations. Sec. 6.3 shows the architecture of the algorithm of EDFLOW. Sec. 6.4 explains the hardware implementation details and Sec. 6.5 shows the experimental results. Sec. 6.6 discusses our results and concludes the chapter.

HW implementation	Aung *et al.* [189]	Huang *et al.* [190]	Haessig *et al.* [191]	BMOF [114]	This work
Algorithm	LP	Gradient -based	DS[0]	BMOF	ABMOF (+SFAST)
Hardware	DVS→ FPGA	Celex→ FPGA	DVS→ Truenorth→ CPU	USB DVS→ FPGA	DVS→ FPGA
Latency us/event	0.36	NA	NA	0.22	1 (0.06-0.1)
OF radius[1]	2	1	1	4[2]	21
Denoising[3]	✓	×	×	×	✓
Keypoints	×	×	×	×	(✓)[4]
AEE (px/s)	99 ± 76[5]	NA	58 ± 53[5]	NA	13 ± 16

[0] Uses 4 Barlow-Levick DS cells [192]; OF computed by CPU. Shown accuracy is based on time-of-flight DS from [115].
[1] The radius of neighbors considered for each OF output.
[2] [114] used only binary event slices, 9x9 blocks, and HD with 1-pixel search distance.
[3] [189] denoises with a local spatio-temporal correlation check [117] and imposes a refractory period before LP OF. [114] denoises with keypoint detector and minimum event density check in OF.
[4] Keypoints are optional.
[5] From [142] using models from [115] and evaluated on sequence *slider_hdr_far* from [1]. [189] and [191] reported accuracy only on a simple rotating bar.

TABLE 6.1: EDFLOW compared with prior DVS OF hardware implementations.

6.2 HARDWARE OPTICAL FLOW

Most current event-based OF methods (see 2.1.2) are based on the traditional Von-Neumann architecture and they are implemented on CPU or GPU, consuming at least tens of watts and large areas of expensive silicon. Computing OF quickly, close to the sensor maintains the efficiency and low-latency advantages of event cameras. Hardware implementations of BMOF for standard video have been claimed to compute more accurate OF than LK methods when the hierarchical matching results are regularized [193], but most implement LK or other nearest-neighbor estimation [194–203], See [193] for a useful summary of hardware BMOF for conventional video OF.

Neuromorphic vision chips over 1986-2000 implemented various models of DS cells from biology [204–208].

Neuromorphic processing ASICs have also been used to compute OF. In 2018, Hassig *et al*. [191] implemented a spike neural network that modeled the Barlow-Levick DS neuron in biology using a DVS and IBM TrueNorth system. However, software are still required to interpret output spikes. The other problem is the accuracy of DS method is poor, like the original hand-crafted feature-based methods.

[135, 209] are the SNN methods for event-based optical flow. and [209] is adapted from Ev-flownet [121, 210].

Fei [128] is another work based on LP fitting. The most important contribution of this work is that they proposed an optimization method based on Prim's algorithm to find the optimal event sets for plane fitting and thus improve the accuracy. They simplify the original SAE model [120] by imposing more constraints on the incoming event and make it a non-iterative. A brief introduction of how to implement it on hardware efficiently is also reported in this work. However, it cannot solve the essential problem that the original LP faced which is that the result is still the normal flow and still suffers from the aperture problem.

Table 6.1 compares EDFLOW with other DVS OF implementations that directly output the flow vectors in pixels per second, including our first BMOF implementation [114]. The LP method was implemented in FPGA in [189]. This work cleverly uses LUTs to avoid matrix inverse and divides the 5×5 plane into 9 subplanes for parallel 3×3 plane fitting with timestamp outlier checking. They report that they can process up to 2.75M events per second. However, the accuracy is also limited by the 3×3 planes and only a 5×5 area around each event, which is much smaller than the 43×43 total search area of EDFLOW. For example, [142] showed that on the *slider_hdr_far* scene from [1], LP produced AEE of 99 px/s which is nearly 8 times more error than dense ABMOF (Table 6.1).

Pivezhandi *et al*. [211] uses the delta value between the adjacent timestamps to compress the timestamp array and save memory. The background activity filter is also included to remove the noise. Their circuit does not compute OF, but rather its input, a 2D event histogram using the constant duration method, which loses many advantages of activity-driven sampling of DVS.

Huang *et al*. [190] used the Celex camera and added a part called Pixel Rendering Module (PRM) to the existing DVS pixel circuit. The PRM helps communication between the center pixel and its 4 neighbour pixels, so it can quickly output the DVS event timestamp and logarithmic intensity for these 5 pixels. Thus they easily can compute local gradients and OF on the

attached FPGA, however, they only provide a couple of figures showing simple flow and patent applications [212, 213]. The PRM requires 5 times the sensor output bandwidth, and estimating the gradient from only 4 neighbours is sensitive to noise. And the prototype is really small, only have 64x64. It is difficult for large-scale applications.

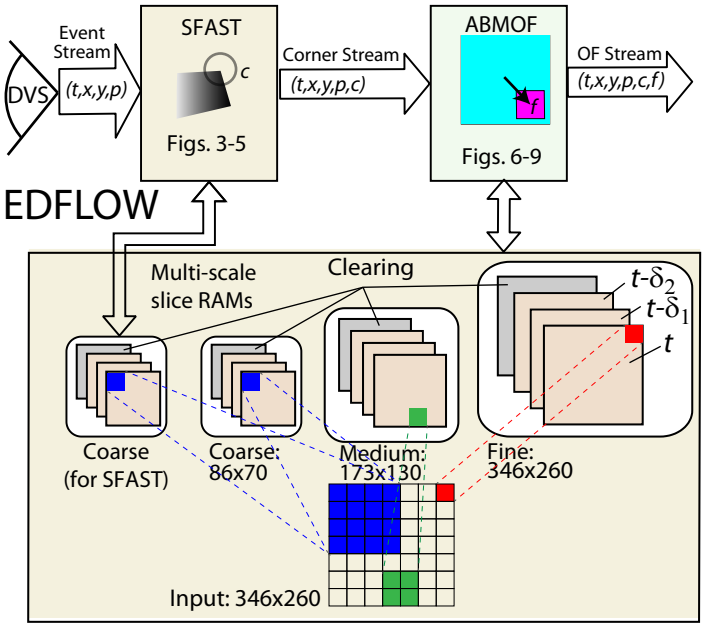


FIGURE 6.2: Top level diagram of EDFLOW SFAST corner detection and ABMOF implementation.

6.3 THE ARCHITECTURE OF THE EDFLOW ALGORITHM

Fig, 6.2 is a top-level diagram of EDFLOW. ABMOF algorithm [142] is a semidense method that computes the OF at points where DVS events signal that the brightness has changed. The event representation used in ABMOF is the *event slice*.

By analogy with photography, we define the ***exposure*** of an event slice as the event accumulation time between the start timestamp and the end timestamp. To make the chapter self-explained, we first give a brief wrap-up of ABMOF and then show the details of SFAST.

A brief summary of ABMOF. ABMOF uses a BM algorithm inherited from video compression for OF estimation. A *block* is a square region in the event slice. The method defines two blocks and uses the similarity metric SAD of the pixel values between the two blocks to compare them. Thus, the OF problem is posed as finding the displacement vector that best matches (by minimum SAD) a Reference Block (**RB**) centered on the event location to a Target Block (**TB**) within a target Search Area (**SA**). Fig. 6.12 on page 122 illustrates these blocks. It is assumed that the appearance of event slices does not change significantly for short times. A multiscale search follows a CTF trajectory to find the best matching TB, following a simple version of displacement estimation by hierarchical BM [214].

The most important contribution of ABMOF is the adaptive slice exposure, which uses the *area event count* method (See Sec. 3.2.2.1) as a feedforward slice exposure controller and the average block matching distance d_{avg} as a feedback controller on the slice rotation number k (See Sec. 3.2.2.2). Together, they dynamically control the inter-slice time interval δ_t to optimize the slice feature quality. These capabilities makes ABMOF robust to dynamic scenes with varying motion speeds and scene structure. Slice FPS can vary from less than 10 FPS for slow surveillance scenes to over 1 kFPS for fast mobile robotic scenes. This feature is important because BMOF methods have a dynamic range of speed that is determined by the slice interval and multiscale search distance. And the area event count method—in contrast to counting the total number of events—makes the slice more invariant to the spatial density of scene contrast [142] and its computational cost is practically zero compared to DNN-based [215] or iterative consistency methods [216]. Since block matching requires only adders rather than expensive multipliers, it is suitable for accelerating on parallel hardware such as FPGA. For more details of the algorithm, we refer readers to Chapter 3.

For detecting keypoint corners, we use a novel method called SFAST, which is adapted from FAST. In this section, we first introduce why we propose SFAST in Sec. 6.3.1. The details of the SFAST algorithm are shown in Sec. 6.3.2 and the difference between EFAST and SFAST from the software aspect is presented in Sec. 6.3.3. SFAST and ABMOF parameters that are used in this chapter are summarized in Table 6.2.

Symbol	Description	Value (default)
$w \times h$	width × height of pixel array	346x260
f_{clock}	clock frequency	100 MHz
(v_x, v_y)	OF result	px/s (pps)
SFAST keypoint detection (Sec. 6.4.2)		
r_{in}	Inner circle radius for SFAST	1
r_{out}	Outer circle radius for SFAST	2
sk_{in}	Inner streak length for SFAST	3-7
sk_{out}	Outer streak length for SFAST	3-11
KP_{thr}	min. ev. count diff. for streak	1-15 (3)
ABMOF block matching (Sec. 6.4.3)		
b_c, b_m, b_f	dimension of course,medium,fine blocks	7,13,25
r	search radius for ABMOF	3
(δ_x, δ_y)	OF displacement vector	±21 pixels
$\text{B}_{\text{sparsity,max}}$	maximum sparsity of blocks	10%
$\text{SAD}_{\text{overlap,min}}$	minimum block overlap for SAD	10%
SAD_{max}	maximum SAD value (% of possible)	50%
d	block match distance $\sqrt{\delta_x^2 + \delta_y^2}$	0-29.7
d_{avg}	average match distance during last slice	
ABMOF area event slice exposure (Sec. 6.4.3)		
k	area event number	k_{min}–k_{max} (700)
k_{step}	step change ratio of k	$1 \pm 1/8$
$k_{\text{min}}, k_{\text{max}}$	min,max k	100,2000
a	area dimension subsampling	5 bits (32x32)
s	# slice scales	3
g	# bits for slice counts	4
δ_t	inter-slice interval between δ_1 and δ_2	
$\delta_{t,\text{min}}, \delta_{t,\text{max}}$	min,max slice exposure durations	1 ms,100 ms
d_{targ}	target match distance for adaptive k	6.3

TABLE 6.2: EDFLOW symbols, description, and values.

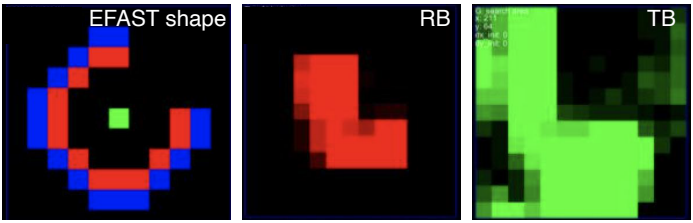


FIGURE 6.3: EFAST successful case. From left to right is the shape of EFAST, RB and TB on coarse scale.

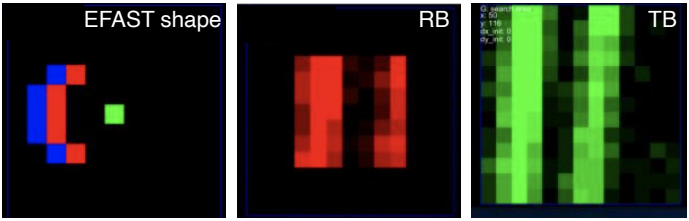


FIGURE 6.4: EFAST failure case. From left to right is the shape of EFAST, RB and TB on coarse scale..

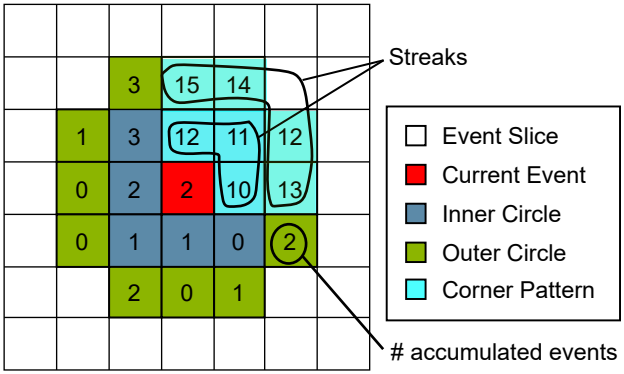


FIGURE 6.5: This example shows a corner event detected by SFAST. The circle radius of 1 and 2 are evaluated on the coarse-scale event slice. See Sec. 6.3.2 for explanation.

6.3.1 *Why SFAST?*

In Chapter 4, we introduced EFAST and its hardware implementation on FPGA. While combing ABMOF and EFAST together, there are some prob-

Algorithm 2: SFAST software algorithm (part)

Data: current event e, threshold KP_{thr}, slice memory $slice[2][346][260]$
Result: corner check result $foundStreak$

```
1  x, y, ts, pol ← e;
2  slice[pol][x][y] ← slice[pol][x][y] + 1;
3  foundStreak ← false;
4  InnerCircleLoop: for i ← 0 to 8 by 1 do
5      for streak_size ← 7 to 2 by 1 do
6          if streak boundary < neighbors then
7              continue
8          minStreakVal ← minimum of streak;
9          maxNonStreakVal ← maximum of non-streak;
10         didBreak ← false;
11         if minStreakVal < maxNonStreakVal − KP_thr then
12             didBreak ← true;
13             break;
14         if !didBreak then
15             foundStreak ← true;
16             break;
17     return foundStreak;
```

lems. EFAST operates on TI but ABMOF operates on *event slice*. As a result, some events might be considered to be "fake" corners in EFAST. Fig 6.3 and Fig 6.4 shows a successful case and a failure case of EFAST. In these two figures, the left parts of them show the block shapes of the current event on the TI. The middle parts of the figures are RBs on the coarse scale of the ABMOF slices. The right parts are TBs on the coarse scale. If we only check it on TI with EFAST, it is obvious that both of them are clear corners. However, Fig 6.4 shows a failure case. On TI, it is indeed a corner, but it is not a real corner on ABMOF *event slice*. On the ABMOF slices, they are actually two parallel bars, and these patterns would cause an aperture problem. To solve this problem and inspired from the fact that both FAST and ABMOF are patch based methods, we proposed a method called SFAST.

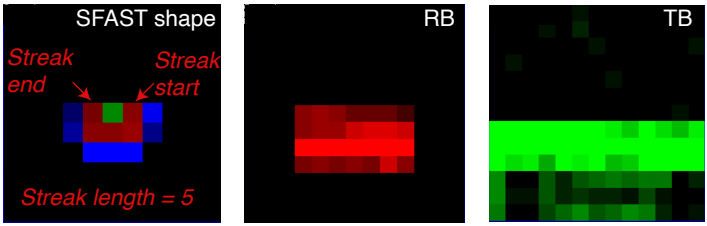


FIGURE 6.6: SFAST false positive corner detection that is prevented by further checking, showing the shape and the slice RB and TB.

6.3.2 *SFAST algorithm introduction*

SFAST is adapted from FAST. FAST is a popular corner detection method in frame-based computer vision because of its robustness and speed [217]. To use FAST on event-based data, we accumulate an event count image in the event slice and treat it as an intensity image. Because both SFAST and ABMOF are block-based methods, we can use the same event slice generated by ABMOF to detect corners. However, to simplify the design and increase the bandwidth, we use a separate copy of the event slice as a dedicated memory for SFAST. Since SFAST only operates on the coarse scale, this extra copy needs only a few BRAM blocks. The steps following the conclusion of the exposure of an event count image are similar to FAST, but we improved the robustness for the relatively low precision of event count images, whose 4-bit counts are quite quantized compared to conventional 8-bit gray scale images.

Fig. 6.5 illustrates SFAST. It shows the slice pixel array of accumulated event counts neighboring the current event. Since DVS event ON and OFF polarity is ignored, the event count is a 4-bit unsigned saturating value. The red pixel in the center is the current event. To detect a corner, SFAST checks both the *inner circle* and *outer circle* centered on the current event. The cyan streaks on the inner circle and the outer circle mean they have the highest event counts in the corresponding circle. If this *streak* is found on both circles, the current event is called a *corner event*. This pattern should satisfy several conditions.

1. The minimum event count of these streak pixels should be sufficiently larger than the maximum of the rest of the count values in the same circle. The gap between the minimum of cyan streaks and the maximum of the rest of the values is called *thr*. Here the length of

the streak or *streak_length* can range from 2 to 7 for inner circle and 3 to 11 for outer circle.

2. If Condition 1 is satisfied, then we might get several streaks with different *streak_length*. Among them, we start from the streak with maximum *streak_length*. This process will check the relationship between the start position of the streak and the *streak_length*, to ensure the streak is valid. The relationship checked here is that if the *streak_length* is an even number, the coordinate difference between the streak start position and the streak end position should be bigger than 2. For streaks of odd *streak_length*, the coordinate difference should be bigger than 1.

The further check in Condition 2 removes some corners that are falsely detected from moving edges. An example is shown in Fig. 6.6. This edge produces a streak that otherwise satisfies the conditions, but it is not really a corner. The additional checks remove corners whose two ends are co-aligned as shown here.

The pseudocode of SFAST is shown in Algorithm 2.

6.3.3 *Differences between software SFAST and software EFAST*

SFAST has a similar idea with EFAST which means that it also does not require to compare the difference between the circle pixels and the center pixel. One crucial difference is the “intensity" images they are operating on. For EFAST, the “intensity" image is the timestamp slice, while for SFAST, it is the event slice generated from ABMOF. The corner checking operation is then performed on ABMOF slices directly. Thus, it is not necessary to create an extra 2D timestamp surface/image to detect corners. This is very useful for hardware because the timestamp surface consumes a lot of memory. After all, the timestamp for every pixel is usually 32bits or more. But for SFAST, according to our experiment, 4 bits per pixel is enough for most cases. The other advantage is that checking the streak directly on the slice is more robust. We also found that in some cases only one circle instead of two circles is enough. And this circle’s radius is 2, which is even smaller than the inner circle of EFAST. This could help reduce the hardware resource further since fewer pixels on one circle; fewer comparators are needed. But to make it more general, we implemented two circles checking in our final implementation. The final change is that no threshold is used in the EFAST method, meaning that there is no hyper parameter.

But in SFAST, we introduced a threshold value as in the normal FAST. This threshold (KP_{thr}) is used to determine if the minimum of the streak pixel values (M) is larger enough than the maximum of the non-streak pixel values (m). The relationship between M, m and KP_{thr} is: $M - m > KP_{thr}$. In EFAST, KP_{thr} could always be treated as 0. The other difference is the further check (Condition 2) in SFAST. EFAST does not have this checking.

6.4 EDFLOW HARDWARE IMPLEMENTATION OF ABMOF AND SFAST

Fig. 6.2 shows the top level EDFLOW diagram. It has 4 major IP blocks: DVS, SFAST, ABMOF, and multiscale RAMs. SFAST accepts the normal DVS event stream as input, writes the SFAST and ABMOF slices and output events that have been detected as corners. ABMOF estimates OF on this corner stream and outputs the final OF stream. Sec. 6.4.1 describes the multiscale event slice accumulation and rotation method, Sec 6.4.2 describes implementation of SFAST, and Sec. 6.4.3 describes the implementation of ABMOF.

6.4.1 *Multiscale slice event accumulation*

Events are accumulated in the current slice t, while slices $t - \delta_1$ and $t - \delta_2$ are used for corners and OF. For simpler logic design and concurrent access, SFAST duplicates the coarse slice RAM. The cost is low since it is the coarse scale. A spare slice of each scale is cleared by a dedicated state machine while the t slice accumulates events. ABMOF requires all 3 scales, but SFAST only uses one coarse scale. There are thus a total of 16 slice RAM blocks.

The slice RAM memory controller increments the slice pixel values in the current slice at all 3 scales. For scales 1 and 2 (medium and coarse), the x and y addresses are right shifted by 1 or 2 bits for subsampling. DVS ON and OFF event polarity is rectified (*i.e.*, ignored) in this design based on our earlier study [142]. To make the blocks have about the same size in the DVS pixel address space among 3 scales, different block dimensions b_c, b_m, b_f are used, listed in Table 6.2. SFAST only checks for corners on the coarse scale, because this scale accumulates the most events and provides the most robust, large-scale corners. ABMOF then uses all three scales following a CTF search to find the best OF vector. Each coarser OF result is used as an initial value of the next finer scale search. Before calculating the final OF, ABMOF performs density checking (Sec. 6.4.3) on all three scales,

which guarantees both the RB and TB are not too sparse and have sufficient overlap. Only after all three scale OF results pass density checking is the summed OF vector from each scale multiplied by 1, 2 and 4 to form the final OF vector, which is output along with the DVS event.

Hardware memory slice reset. Besides the memory layout, the other difference between the software implementation we did in jAER and this hardware implementation is in the memory slice number. On the software side, we use 3 slices which are slice t, slice $t - \delta_1$ and slice $t - \delta_2$. When an incoming event triggers a slice rotation in the software, the $t - \delta_2$ slice will be first cleared, and then it is set as the new slice t to accumulate events. That operation is straightforward in software and memory clearing operations are highly optimized on CPUs. However, using hardware to reset a whole memory slice is a time-consuming task. For example, if we use a dual RM port memory and adopt the optimized memory layout as we have shown in Sec. 4.3.3, it takes about 4086 cycles to finish the reset operation. This means that during these 4086 cycles, it is not possible to accept a new event. For a system running at 100MHz, it is around 41us. The temporal resolution of the camera is 1us, according to the estimate maximum event rate 20M event per second (Sec.5.3.3), 41us might skip more than 800 events. Therefore, we add an extra memory slice. This slice is cleared by a dedicated state machine while the other slices are being processed (accumulating or reading data from memory slices). Once a slice rotates, an empty slice has already been prepared and no reset operations are required before accumulating. This hardware saves a lot of time for slice rotation.

6.4.2 *SFAST hardware keypoint detector*

Fig. 6.7 shows the workflow of the hardware SFAST. The input of the system is the event stream. Event stream means the raw data stream. The stage corner stream is an intermediate result stream. It first checks the inner circle. If the inner circle satisfies the corner condition, then it checks the outer circle. Otherwise, it outputs false immediately. Only it passes both stages checking, it reports a true corner. No matter in inner circle or outer circle checking, SFAST tries to search the specific pattern on the circle. The output is the maximum streak length. If it is less than 3, then it is not a corner.

The SFAST corner check consists of a slice memory controller, a sorting unit, and a contiguous index detector. The first part accumulates incoming

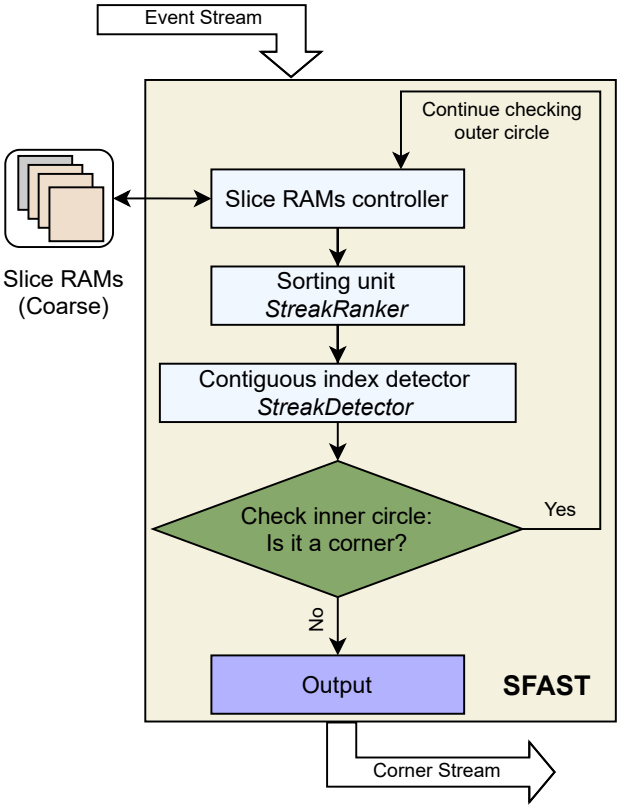


FIGURE 6.7: The workflow of the hardware SFAST corner detector. See Sec. 6.4.2 for explanation.

events to the slices and reads the SFAST circle data from the slice. The second and third part are designed as two PEs: *StreakRanker* and *StreakDetector*. The principle and circuit of these two PEs are explained in this section.

Considering that the only difference between inner circle and outer circle checking is the circle radius, we designed a general circuit to save resources.

First, *StreakRanker* sorts the accumulated counts to a list of event count ranks; *i.e.*, each entry of the list is the rank of event counts, with the largest rank corresponding to the pixel with the largest event count. Then a second circuit *StreakDetector* detects streaks of the pixels with the largest event counts by detecting the longest contiguous set of ranks starting from the

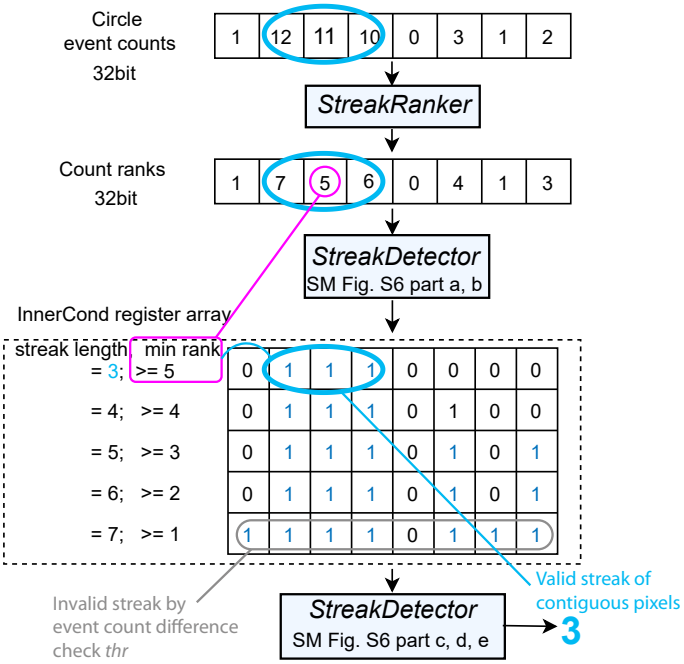


FIGURE 6.8: Simplified architecture of the SFAST *StreakRanker* and *StreakDetector* blocks that detect a keypoint streak on the FAST circles. For SFAST, the input is the array of event counts on inner or outer ring. The example shows a streak consisting of 3 pixels. *StreakRanker* returns the rank of the corresponding input data in the sorted list. *StreakDetector* first returns an inner condition array which stores the comparison results for different streak lengths and minimum accumulated event count. The maximum length of the streak which has enough contiguous pixels is returned by *StreakDetector*. In this example, the data marked by a blue ellipse is a streak with the maximum streak length that satisfies the corner condition. Its minimum count ranking is 5, which also satisfies the condition.

largest rank. This contiguous set represents a potential streak: If all event counts for this set are sufficiently large, then it is regarded as a streak. Fig. 6.8 shows an example to illustrate the detection of the streak on the inner circle of Fig. 6.5. The data list on the top is the circle data but unrolled as an linear data list for better illustration, but is implemented as a circular buffer; the last data of the line is the neighbour of the first data in the

list. This data list is read by *RWSlices* from the coarse event slice $t - \delta_1$. *StreakRanker* returns the rank of the corresponding input data in the sorted list and packs the results in a wide bit depth value. The inner circle has 8 values and every value has 4 bits, so it is efficiently packed as a 32-bit value. Fig. 6.8 shows that the second data of the result is 7, which means that the corresponding entry event count 12 is the maximum value in the input list. The schematic of *StreakRanker* is shown in Fig. 6.10.

Second, *StreakDetector* finds the valid streak with the maximum streak length. All streak lengths to be checked for the inner circle are shown in Fig. 6.8. The check is best explained by an example: Suppose the check is for 3 contiguous largest counts on the 8-pixel circle, and the ranks of these 8 pixels have been computed by *StreakRanker*. The check ensures that all ranks of these 3 pixels are larger than 5 and is done in parallel. Similarly, a check for 4-pixel long streaks only needs to ensure all ranks are larger than 4. Among these valid streaks, *StreakDetector* returns the maximum streak length. If no valid streaks are found, *StreakDetector* returns 0. The schematic of *StreakDetector* is shown in Fig. 6.11.

The circuits of *StreakRanker* and *StreakDetector* shown in Fig. 6.8 check only one data per clock cycle. Therefore, 8 cycles are required for the inner circle and 12 cycles are required for the outer circle checking. If both stages are processed, 20 cycles are required. To decrease the latency, we used 4 copies to parallelize the check, to process 4 data per cycle and so the check requires at most 5 clock cycles. Most events are not corners, and thus require only 6 clock cycles, resulting in a peak throughput of 16.6 Mevents/s.

6.4.2.1 *Schematic of StreakRanker and StreakDetector*

When dividing wide-bit data into several narrow-bit data and then outputting them one by one in order, such as converting a 32-bit to 8 4-bit data and outputting them from the Least Significant Bit (**LSB**) to MSB, a typical circuit is shown in Fig. 6.9. In Fig. 6.9, the counter part generates a control signal and increases itself 1 every clock. This control signal is used to select one 4-bit data output from all the eight 4-bit data inputs. The resource consumed here is an adder, a Flip-flop and a MUX. Actually, a more efficient circuit could be designed as the data is output one by one in a specific order. This circuit is shown in Fig. 6.10 part a. It uses a 4-bit cycle right shifter to shift the input data 4 bits every cycle. In this design, only a 4-bit cycle shifter and a Flip-flop are required. It is worth noting that shift operation on the hardware does not consume any resources. By using this cycle shift read method, we can avoid using multipliers that consume a lot

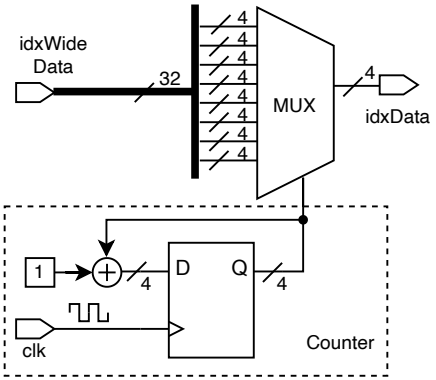


FIGURE 6.9: The principle of PE *StreakRanker* part a.

of LUTs. This circuit is also used as the input part circuit of *StreakDetector* in Fig. 6.11. The output of *StreakRanker* (Fig. 6.10 part c) adopts the similar idea to save the resources.

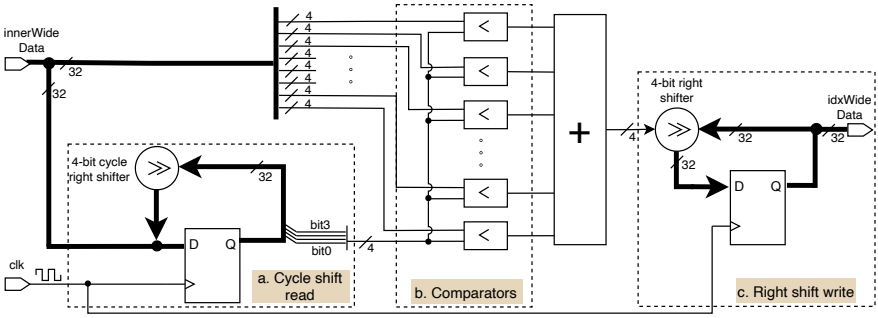


FIGURE 6.10: The circuit of *StreakRanker*. *Inner circle* is used here as the example. It consists of three parts. Part *a* reads the input 4-bit data one by one. Comparators compare the input 4-bit data with rest of the data. The last part *right shift write* is for output.

As shown in Fig. 6.10, we use *inner circle* as an example here to show how it is implemented on hardware. This PE consists of 3 parts: Cycle shift read, comparators and right shift write. The input of this PE is a wide data consisting of 8 values since the *inner circle*'s size is 8. Because every data has 4 bits, the bit width of the input is 32 bits. Cycle shift read reads the

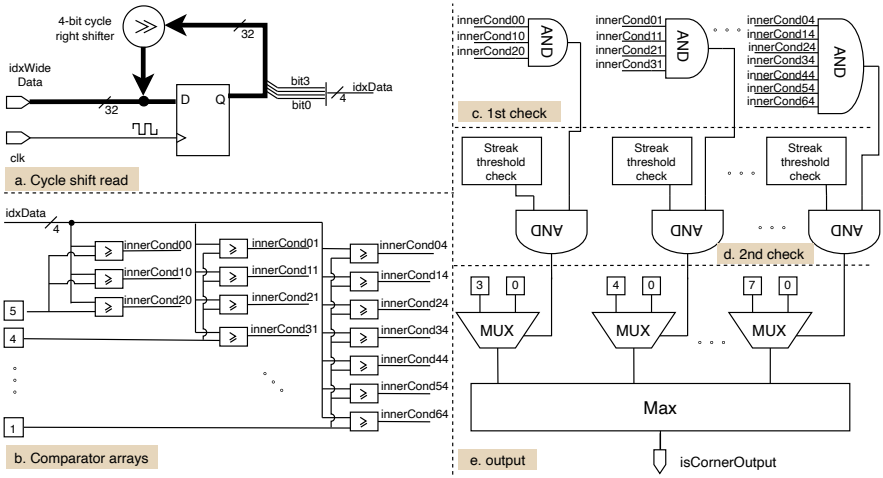


FIGURE 6.11: The circuit of *StreakDetector*. *Inner circle* is used here as the example. It consists of 5 parts. Part *a* is for cycle reading and reads input. Part *b* converts the index to a binary list. Part *c* checks if there is an uninterrupted streak. Part *d* further checks if the threshold condition is satisfied. Part *e* returns the satisfied streak with maximum streak length. Otherwise 0 is returned.

least significant 4 bits from the wide data input. After reading, the input data is shifted right 4 bits. This shift makes sure that in the following clock cycle, the next data is read. The input data is then sent as a common input of 8 comparators. Other inputs of these 8 comparators are the whole 8 data. If the input data is smaller than the rest data, the comparator returns 0. By summing all these results together, we can get the rank of these data in the whole data list as we shown in Fig. 6.8. The output of this PE is the ordered index of the input data list. To make the next PE be able to use cycle shift reads rather than multipliers, we add a right shift write part to the output of the result in a wide data format.

StreakDetector: Fig. 6.11 shows the circuit that detects streaks from the ranked accumulated event counts. This circuit is used for both *inner circle* and *outer circle* SFAST checking. The input of this PE is similar to *StreakRanker*. It also uses the cycle shift read to get the origin data from the wide input data. This data idxData is compared with some constants. The constants range from 5 to 1 and it is corresponding to the streak length range from 3 to 7. Part b generates an Innercond register array. Every col-

umn of this array corresponds to a specific streak length. An example of this column can be found in the bottom list of Fig. 6.8. The register array is then sent to a list of AND gates. The outputs from the same column or same streak length connect to the same AND gate. After the first check, the second condition checks if the maximum of the streak is bigger than the minimum of the nonstreak. The output of the second check is used as the selected signal of the final output part. If the output is 1, it means it is a valid streak and the corresponding streak length is selected, otherwise 0 is selected. By combining the result of these MUXs and a Max module, the streak with maximum streak length is output. If there is no valid streak on the circle, *StreakDetector* outputs 0.

6.4.2.2 *Difference between hardware EFAST and hardware SFAST*

In Sec. 6.3.3, we showed the software difference between EFAST and SFAST. In this section, we will compare them from the point of view of hardware. The whole framework of the hardware SFAST is almost the same as the hardware EFAST except the circle size, circle number and threshold checking. For these differences, no more new PEs are required. Only a few logic on some PEs are required to be modified. However, for the wrong pattern checking (Condition 2 in Sec. 6.3.2), a new PE is created. This PE takes the result of the *StreakDetector* as the input and output the final checking result. The implementation of this new PE consists of a few logic gates and it is not very complicated. Even though one more PE is required for EFAST, but as shown in the previous context, SFAST checks on smaller circles and thus finally the total resources for SFAST are less than SFAST. Besides LUTs, the memory resources consumed decrease significantly. EFAST uses TI which has 32 bits for every pixel while SFAST adopted the ABMOF event slice that has only 4 bits per pixel. It decreases the bit width from about 32 to 4 and thus the whole BRAM consumed on FPGA is about 8 times less than in the EFAST implementation.

6.4.3 *ABMOF hardware design*

Fig. 6.12 is the architecture of the ABMOF hardware IP block. It has three types of modules: PE, RAMs, and FIFOs. Each of the 7 PEs is a small computation unit that has a specific function. The RAMs store the t, $t - \delta_1$, and $t - \delta_2$ event slices, each at 3 spatial scales. In order to use the memory in a more efficient way, the algorithm used a mechanism called slice rotation to make the memory could be reused. The FIFOs buffer data

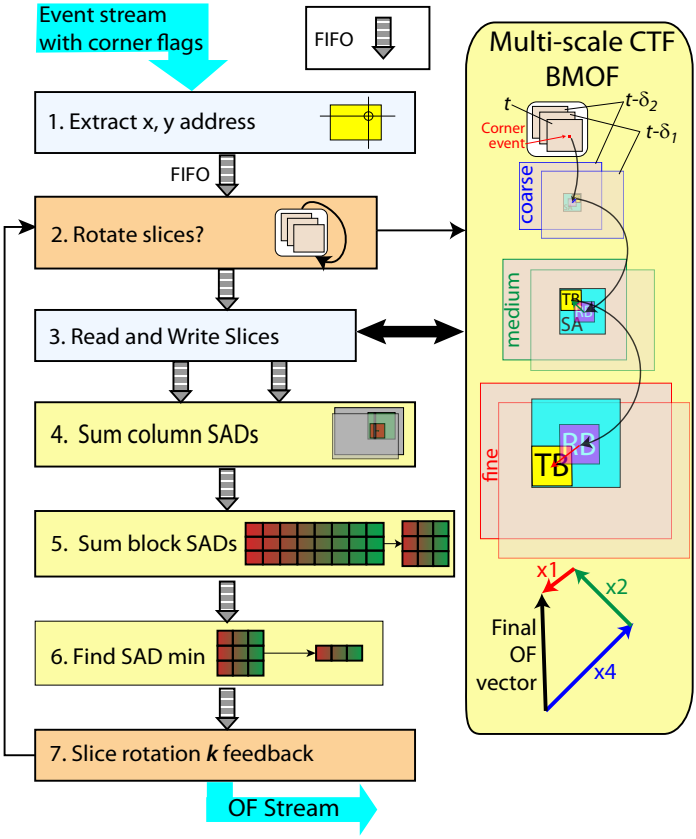


FIGURE 6.12: Adaptive block matching optical flow PEs (ABMOF).

to synchronize between PEs with different data rates. Due to different PEs have different initial latency, all PEs cannot be connected directly. To solve this problem, FIFOs are added between every two PEs and make the whole IP work in a consumer-producer way. The faster PEs will stall until valid data is generated from the slow PEs. In this way, PEs having different initial latency could be synchronized and work in parallel. All PEs are controlled by the same clock source, which is the global logic clock running at 100MHz.

1. ***DVSInterface*** receives and parses AER data from the DVS and writes a FIFO that buffers variable rate DVS events.

This IP makes an interface conversion over an AXI4 Stream (**AXIS**) protocol. AXIS is a communication protocol by ARM. The simplest form of AXIS is similar to FIFO and includes three mandatory ports: TVALID, TREADY and TDATA. TVALID means the data is valid and TREADY indicates that the receiver or slave is ready for new input accept. TDATA and TVALID are inputs and TREADY is the output for AXIS slave. The data can be read by the master only when both TVALID and TREADY are high. The full AXIS protocol has 4 other extra optional ports: TUSER, TLAST, TKEEP and TSTROBE. These 4 optional ports are not used in this IP. For details about AXIS, we refer readers to [218]. The reason we use AXI4 as the internal data protocol for all the IPs in this platform is because of its simplicity and address-free feature which can save a lot resources. It is also a very natural model for events since events are generated in a sequence like a stream.

The other function of this module is to filter out boundary events. Because for every event, we require a certain area around this event for processing. To remove the bias effect, events that are very close to the boundary is removed in this PE. PE 1 also filters out events that are too close to the boundary for OF computation.

2. ***SliceRotator*** generates the rotation flag signal to the slice memories using the area event count method, using feedback from PE 7 that controls the area event number k [142]. It divides the whole 346x260 fine-scale event slice to 12 by 9 32×32-pixel blocks; if the total event count of any block exceeds k, it triggers the slice rotation. k is changed by fixed steps to control d_{avg} towards d_{targ}. We impose hard limits $k_{\text{min}} \leq k \leq k_{\text{max}}$ and $\delta_{\text{t,min}} \leq \delta_{\text{t}} \leq \delta_{\text{t,max}}$. $k = 100$ roughly equates to 1 event per 3×3 pixels in the area. The most important function of this PE is to read the column data from the RAMs and update the latest RAM slice according to the current event's location and polarity.

3. ***RWSlices*** is a slice RAM controller. The dataflow directive in Xilinx design environment Vivado HLS requires that global values such as global registers or memory should only be written or read in one PE. By encapsulating the memory IO in *RWSlices*, we avoid the competition caused by several PEs trying to access or update the same memory location simultaneously. PE 3 updates the current event slices at each scale for every incoming event. As shown in Fig.6.13, *RWSlices* reads out the data from the slice $t - \delta_1$ and slice $t - \delta_2$ and then stores them in the internal register array as RB and SA. *RWSlices* outputs the data in two FIFOs column by column in a specific order illustrated in Fig.6.13.

Block matching and multiscale SAD considerations: The work in Chapter 2 was a basic FPGA implementation of event-based block-matching OF. In that work, the sensor was only 128×128 pixels, the block size was 3×3, and the search distance was only 1 pixel. The total Binary Hamming Distance (**BHD**) operation only consumed 81 gates so we could implement the binary slice memory as registers and the BHD as gates to compute the BHD in a single cycle. By replacing the HMD with SAD and considering that SAD for each pixel would require one subtract and one absolute value selector, it would consume $81 \times 2 = 162$ adders and 81 selectors which is an easy case for a small-scale FPGA. We could unroll it completely and make it possible to get the minimum SAD value in one cycle very easily. However, in this work, we have 3 scales and the block size for different scales is different. For the fine scale, the block size is up to 25×25 and the search distance is also increased to $[-3, 3]$. This means that if we still finish the whole SAD operations in one cycle and unroll it completely, then it will consume at least $25 \times 25 \times 25 \times 7 \times 7 \times 2 = 61250$ adders and 30625 selectors. Here, the slice memories are much larger and use 4-bit event counts and 3 scales. The search distance is ± 3 slice pixels at each scale. Register implementation would consume hundreds of thousands of gates. This consumes hundreds of thousands of LUTs only for the block SAD part. And to calculate the block SAD in one cycle, the two blocks used for calculation would need to be accessed in one cycle. In [114], the slice memory was binary with 128x128 resolution, so we could use flip-flops to build this memory.

Here the slice image is increased to 346×260 with 4 bits per pixel and so we need to use the FPGA block memory. Thus, there is no way to read the whole block in one cycle due to the memory ports limitation. Since the throughput performance of a system is limited by the slowest part, it is not possible to ready the IP for the next input in less cycles than is required to read out all the target block data from the event slice memory. The number of clock cycles before the function can accept new inputs is also called the Initial Interval (II). II is generally the most critical performance metric in any system. II of this hardware ABMOF is determined by the maximum II of all PEs. The hardware design is thus a trade-off between throughput and resources. Therefore, here we used FPGA BRAM, and divided the process to construct the minimum block SAD into 3 subprocesses and designed them as 3 PEs: *ColSad*, *RowSummer* and *FindStreamMin*. For each scale starting from the coarse scale, it reads the RB and SA into register memory and pipelines the SAD computations for the 7x7 SA. By careful partitioning

of the slice block memories, the entire BMOF computation requires only about 100 clock cycles, which is 1 us at our 100 MHz clock frequency. The maximum search distance is ±21 pixels in x and y directions. It enables BM results from large block dimensions equivalent to about 25 × 25 DVS pixels at each scale.

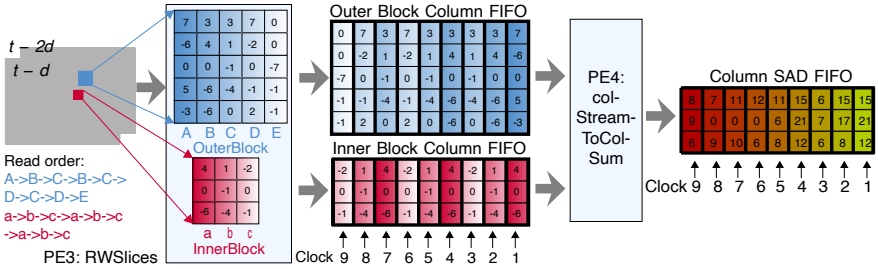


FIGURE 6.13: The column SAD computation of the Fig. 6.12 PEs 3+4. Two gray rectangles here represents the DVS memory slice of $t - \delta_1$ and $t - \delta_2$. The blue SA and red RB are centered on the current event. Here we choose $b = 3$ and $r = 1$ for illustration (the actual values are $b = 7$ and $r = 3$). The first FIFO data is the right column and the last data is the left column.

4. ***ColSad*** returns the column SAD value, *i.e.*, the SAD values for columns from the RB and entire SA. It takes the columns generated by *RWSlices* as input and calculates the column SAD one by one. The calculation process of PE 4 is like this: There are $2 * r + 1$ column SADs, where r is the search radius (Table 6.2). In this example, $r = 1$, so $2 * r + 1 = 3$. Thus *ColSad* accepts 2 input data with size $b = 3$. The first column of the RB is [4, 0, -6] and the first column of the SA is [7, -6, 0, 5, -3]. To calculate the SAD between these two columns, which with different lengths, the second column is split to 3 subcolumns, by shifting from top to bottom: [7, -6, 0], [-6, 0, 5] and [0, 5 ,-3]. *ColSad* outputs the SADs between these subcolumns and the first column. To save LUTs, part of the accumulate operations of *ColSad* are implemented on DSPs. Therefore, 3 output data are generated as shown in the figure as [15, 21, 12]. The rest of the columns are calculated in the same way. These results are input to a FIFO connected to PE 5. Finally, $(2 * r + 1) * b$ columns are generated for one event.

5. ***RowSummer*** (Fig. 6.14) sums the rows of column SADs to compute the final block SAD. The input data stream is the column SADs generated from *ColSad* as shown in Fig 6.13. Every $b = 3$ columns are summed to-

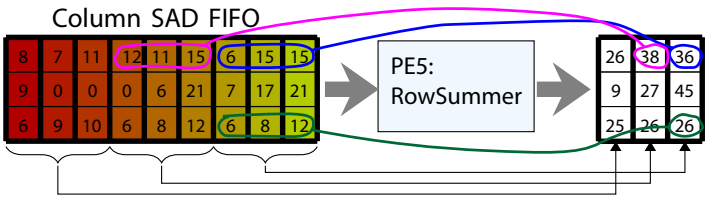


FIGURE 6.14: PE 5 *RowSummer* sums rows of the PE 4 *ColSad* output to result in final block SAD result. In this example, every 3 columns in the same row generate a block SAD.

gether. Therefore, $2 * r + 1 = 3$ data is calculated and every $b = 3$ data makes up a column. The final result is stored in a FIFO with $2 * r + 1 = 3$ columns and $b = 3$ data per column. Data of offset (x, y) on the output array corresponding to a block SAD result between the RB and the block offset with (x, y) in the SA. Block SAD results obtained from *RowSummer* are sent to PE 6.

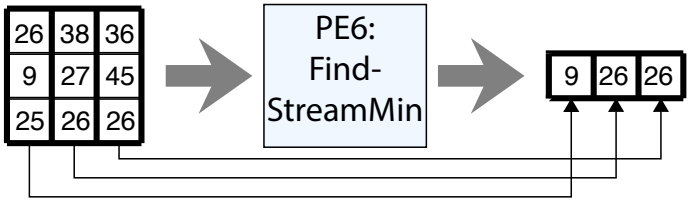


FIGURE 6.15: The principle of FindStreamMin. It reads the block SAD value column by column and return the minimum SAD in each column.

6. ***FindStreamMin***. PE 6 (Fig. 6.15) finds the best matching block with minimum SAD. There is a global minimum stored in a register. On every cycle, *FindStreamMin* reads one column of data from the FIFO, computes its minimum value and compares the minimum with the global register value. It stores the possible new minimum and its offset in two registers. It finally outputs the offset OF result and and SAD for density checking.

Block search: The block search uses a CTF strategy: Fig. 6.12 shows how the search starts at the corner event location in slice t. At each scale the RB from slice $t - \delta_1$ centered on the previous scale best match location is compared with TBs in slice $t - \delta_2$ over the SA. The block size varies over the scale to keep nearly the same 25×25 block size in the DVS pixel array. Each search is an exhaustive full search over the entire 7×7 SA;

many implementations use a sparser search strategy [219] but we found, like [193] that a sparser search strategy such as Diamond Search [143] as we did in software has too many conditional branches, which makes it difficult and inefficient to implement in hardware. For details refer to Sec. 4.3.2. The search can be aborted if either the source or target best match is too sparse. In case the OF search completes, the OF output is a (δ_x, δ_y) OF displacement vector which is the scaled sum of the OF at each scale. The length of this vector is the match distance d.

7. ***DensityCheckAndFeedbackControl***. PE 7 has two functions: First, it checks for sufficient BM feature density. If the final minimum SAD value is larger than the threshold SAD_{max}, or the non-zero area of RB or TB is too sparse ($\text{B}_{\text{sparsity,max}}$), or there is insufficient RB and TB non-zero overlap ($\text{SAD}_{\text{overlap,min}}$), then the OF is flagged invalid. Second, it provides area event number feedback control (See Sec. 3.2.2.1). It calculates the actual average block match distance d_{avg} during the slice rotation period, and compares it with the target d_{targ}. This feedback value E_{d} is sent back to PE 2 to adapt k to the dynamic environment. If the error signal $E_{\text{d}} = d_{\text{avg}}/d_{\text{targ}}$ is larger than 1, k is reduced by factor k_{step}, otherwise it is increased.

Density checking: Density checking is used to check if the RB and the TB are too sparse or have insufficient overlap. It checks three conditions: First, it calculates the percentage of nonzero values of RB and SA and checks if this percentage is lower than a specific threshold $\text{B}_{\text{sparsity,max}}$ (10% in this design). Second, it checks the percentage of their nonzero intersection area. The nonzero intersection area represents that the event counts on both of RB and TB are positive. The threshold $\text{SAD}_{\text{overlap,min}}$ of this check is also set to 10%. Third, it checks that the final minimum SAD value is not larger than SAD_{max}. A sparse event flag signalling invalid OF is set if any if these conditions are true.

To implement the density checking on hardware, one essential part is to count the zero values on RB, TB and their intersecting areas. In software, it is calculated after we get the whole block. However, we use a column-by-column method in this design. We do not have a complete block for counting. Therefore, we proposed a similar way as we do in calculating the block SAD. For every column from RB and TB, we count the number of zeros in the column. These numbers are sent to the next PE along with the column SAD value. Comparison with the threshold parameters ($\text{B}_{\text{sparsity,max}}$ and $\text{SAD}_{\text{overlap,min}}$) is checked in PE 7.

6.4.4 *Unroll trick used in the hardware design to increase parallelism*

As shown in PE 4 (Sec.6.4.3), there are $(2 * r + 1) * b$ columns generated for every event. To maintain the block shape consistency among the three scales, the block dimension is varied. The fine scale has the largest block dimension $b_f = 25$, so it should have the longest II. Substitute b with b_f and $r = 3$, the columns generated by the fine scale is $7 \times 25 = 175$. If only one column is processed every cycle, the II would be more than 175. The maximum II limits the II of the whole system. Our goal is to achieve a system with II around 100. Therefore, we should decrease the II for the fine scale. As we said before, the hardware design is a trade-off between throughput and resources. To increase the throughput (decrease II), more resources would be consumed. Here, we use a pretty common trick called unroll in the hardware design. Unroll would duplicate the hardware and thus increase parallelism. In the fine scale, 175 columns need to be processed. Moreover, they are divided into 7 loops; every loop needs processing 25 columns. If we partially unroll the loop, such as a factor 2, one loop could be finished within $25/2 = 13$ cycles instead of 25 cycles. Therefore, the total cycles would be changed to $7 \times 13 = 91$ cycles. By using the unroll technique, we can make the system satisfy our goal. The loop could be unrolled completely (one loop only requires one cycle), but it means 25 units would be duplicated. The unroll factor should be chosen carefully to balance the resource and II. The unroll factor is also called Number of Parallel Computation (NPC). *StreakRanker* and *StreakDetector* also adopt this unroll technique.

6.5 EXPERIMENTAL RESULTS

To evaluate the performance of EDFLOW, we tested them on the `Davis346Zynq` platform (See chapter 5). We also implemented EFAST to compare its performance with SFAST. Details of EFAST hardware implementation are in Chapter 4. Quantitative and qualitative results on the accuracy and speed-up of processing time are in Sec. 6.5.1 and 6.5.2.

6.5.1 *OF accuracy on baseline dataset*

To evaluate the performance of combining ABMOF with corner detection methods EFAST (**ABMOF+EFAST**) and SFAST (**ABMOF+SFAST**), we first tested it on the baseline dataset DVSFLOW16 using the test sequence *trans-*

box [115], recorded from a *DAVIS240C* camera that is panned over a scene with several boxes. Since the camera is panning around its own axis, the OF ground truth is uniformly horizontal and is obtained from the built-in IMU. The qualitative result of ABMOF and its variants on this baseline is shown in Fig. 6.16 and the quantitative result is shown in Table 6.3. Fig. 6.16 shows that the original ABMOF has the densest output, but it is clear that there is some noise in the result. Both ABMOF+EFAST and ABMOF+SFAST show sparser but cleaner results. To quantify the performance, we use AEE, outlier percentage and event density as the metrics. AEE is a common metric for OF evaluation. The outlier metric was introduced in the KITTI benchmark 2015 [220], which defined it as a flow vector result with an endpoint error larger than 3 pixels or 5% of the Ground Truth (GT) magnitude. We define the event density as the fraction of events that are processed for OF. Table 6.3 shows that the original ABMOF is the most dense method. However, it is still less than 100% because the density checking filters out some events in ABMOF. It rejects the event if the block around the event is too sparse. The accuracy of ABMOF+EFAST and ABMOF+SFAST are almost the same, but ABMOF+SFAST is more dense. It can also be observed from Fig. 6.16. The other thing we can learn from Fig. 6.16 is that corners extracted by SFAST are more concentrated while EFAST are more distributed, meaning there are more “isolated corners". Many of these are the failure cases we show in Fig. 6.4. Generally, we observe that SFAST produces fewer errors.

6.5.2 *OF accuracy on more complicated dynamic scenes*

We also tested EDFLOW on more complex indoor drone flying and outdoor nighttime driving scenes from MVSEC [24]. Here we also compared ABMOF methods with the CNN OF estimation method *Ev-Flownet* [129]. MVSEC has two main scenes: *indoor flying* and *outdoor driving*. MVSEC combines several sensors and capture systems such as stereo frame-based camera, LIDAR, IMU, motion capture, GPS, and a stereo *DAVIS346B*. Although it does not provide OF ground truth directly, it provides OF GT indirectly based on the direct depth and camera pose ground truth data. For indoor flying sequences, the whole background is static and only the drone is moving. In this situation, OF GT is accurate enough. However, for outdoor driving sequences, since OF GT is obtained from camera ego motion, the converted OF GT has some errors on the independent moving objects such as pedestrians or cars on the street.

	AEE/px	% Outlier	% Event density	# Events
		transbox		
ABMOF	0.78	3.31	**55.56**	**137973**
ABMOF+EFAST	**0.73**	**0**	2.84	7059
ABMOF+SFAST	**0.73**	0.76	6.88	17083
		indoor_flying1		
Ev-Flownet	**1.03**	**2.2**	100	
ABMOF	2.04	21.8	67.2	9458979
ABMOF+EFAST	2.04	21.2	1.1	153624
ABMOF+SFAST	**1.90**	**18.6**	4.1	
		indoor_flying12		
Ev-Flownet	**1.72**	**15.1**	100	
ABMOF	3.74	45.4	66.4	16619321
ABMOF+EFAST	4.09	47.9	1.20	301742
ABMOF+SFAST	**2.74**	**33.4**	3.37	845062
		indoor_flying3		
Ev-Flownet	**0.55**	**11.9**	100	
ABMOF	3.06	38.23	**67.58**	**16220988**
ABMOF+EFAST	3.24	40.42	1.02	245729
ABMOF+SFAST	**2.45**	**30.31**	4.41	105932
		outdoor_night1		
Ev-Flownet	**0.49**	**0.2**	100	
ABMOF	2.58	27.1	**69.8**	**60797614**
ABMOF+EFAST	**2.28**	**24.4**	4.5	3959042
ABMOF+SFAST	2.47	27.5	5.9	5161382

TABLE 6.3: ABMOF and its variants accuracy comparison.

Fig. 6.17 shows the qualitative result on *outdoor_night1*. All ABMOF variants reproduce the expanding GT flow field, but the ABMOF+SFAST detects more keypoints than ABMOF+EFAST. Quantitative results of all indoor flying sequences and outdoor driving sequence are also reported in Table 6.3. The accuracy of *Ev-Flownet* is taken from [129]. It is not surprising that *Ev-Flownet* has the best accuracy. ABMOF+SFAST achieves the best AEE accuracy of the ABMOF methods for almost all situations except in the outdoor flying sequence. Without corner filtering, ABMOF is obviously the most dense method. Comparing only ABMOF+SFAST and ABMOF+EFAST, ABMOF+SFAST detects more corners and the corners are more concentrated, and ABMOF+EFAST has more noise corners. This observation is the same as in the baseline result. Detecting more corners is useful if the OF is used as a base method for higher-level applications such as VOD or SLAM because it can help build a more dense map.

There are several reasons for the accuracy gap between the ABMOF series methods and *Ev-Flownet*. The main ones are that the CNN method can tune hundreds of thousands more parameters and thus better handle complicated scenes and nonidealities. Moreover, the multilayered convolutional CNN can compute consistent, smooth OF and thus have far fewer outliers.

6.5.3 *Adaptive slice exposure control*

As discussed in Secs. 6.3 and 6.4.3, ABMOF controls the slice exposure using the rotation area event count number k, which can itself be put under feedback control to center the average BMOF matching distance in the range of possible match distances. We observed that the feedback control of k does not always produce a correct k value. For example, if k is too large, then the slices are "overexposed" and have a lot of motion blur, or the displacement can exceed the maximum search distance. Even if the displacement is not too large, the motion blur results in poor BMOF. Likewise, if k is too small, then the slices are "underexposed" and provide insufficient event density for matching, or the displacement is so small that the flow is excessively quantized. The hard limits on k and δ_t help ensure that the exposures result in meaningful flow, but we sometimes find that it is better to fix k.

Using adaptive k is useful when speed varies a lot over time. Fig. 6.18 shows an experiment with a rotating dot covering a speed range of more than 2 decades. Feedback control was activated. Fig. 6.19(a) shows AB-

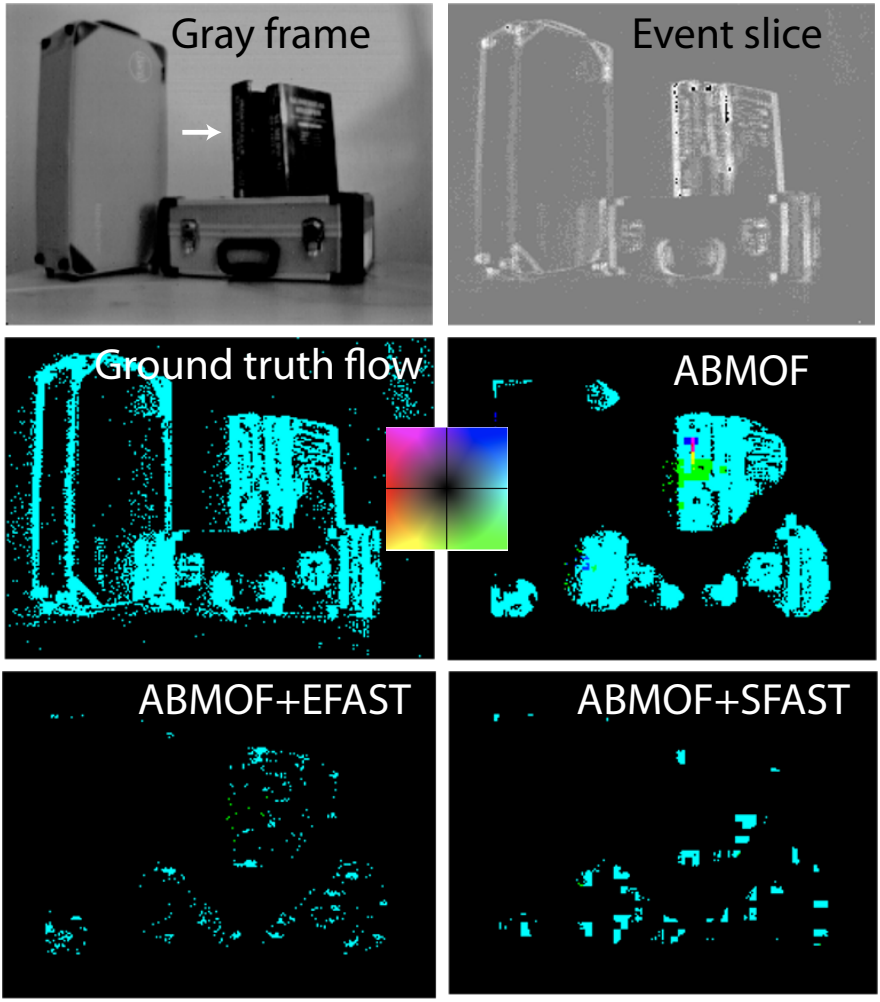


FIGURE 6.16: Snapshot of 40 ms of OF from ABMOF and its variants on *trans-box* [115]. Color wheel shows OF direction. It is also applicable to Fig. 6.17. Color brightness indicates speed.

MOF quantities. We can observe that as a result of feedback control, the k is controlled starting from an initial value of 1 kev (thousand events) down to about 300 ev as the dot speeds up. We can see that the feedback control results in an average matching distance d_{avg} that stabilizes around $d_{\text{targ}} \approx 7$ px. Fig. 6.19(b) plots k and δ_{t} versus the dot speed. We can ob-

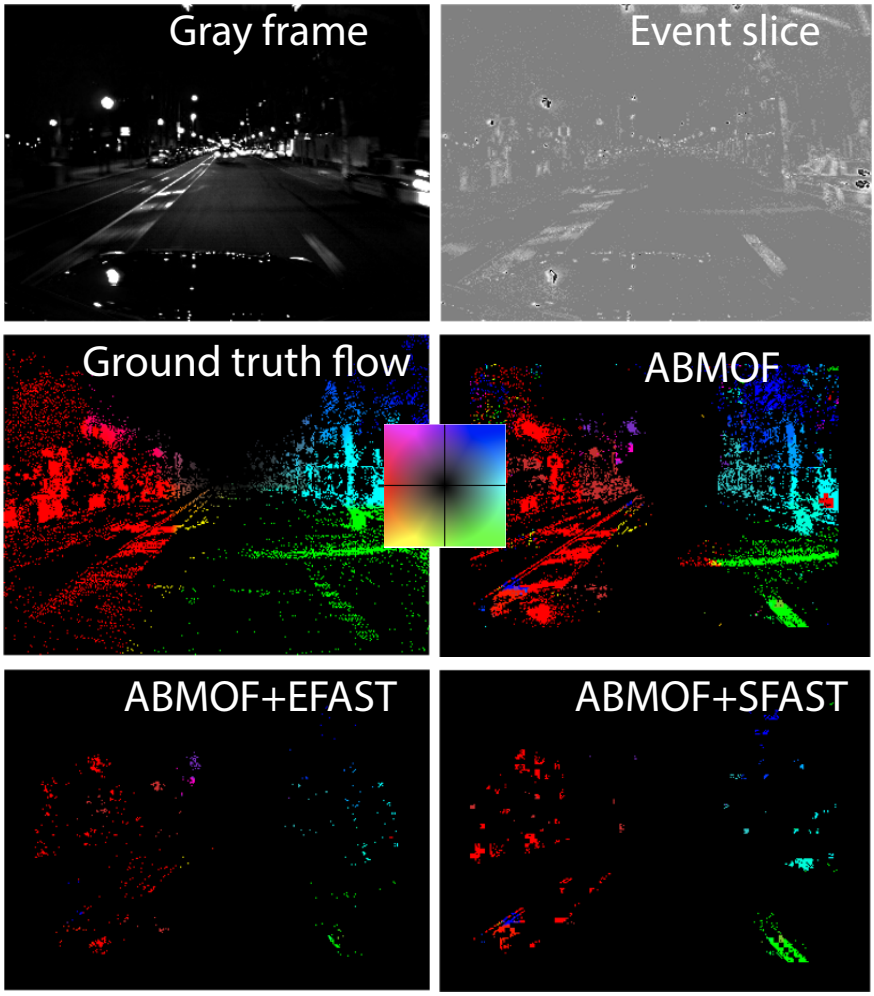


FIGURE 6.17: ABMOF and its variants ABMOF+EFAST and ABMOF+SFAST on *outdoor_night1* from [129]. Best viewed in color.

serve that using the feedforward area event count makes δ_t vary inversely with dot speed over most of the speed range. The k feedback control makes k decrease approximately as the square root of dot speed, *i.e.*, it makes δ_t decrease faster with speed more than it would without feedback control (where k is fixed). Fig. 6.19(c) shows that using adaptive k improves the

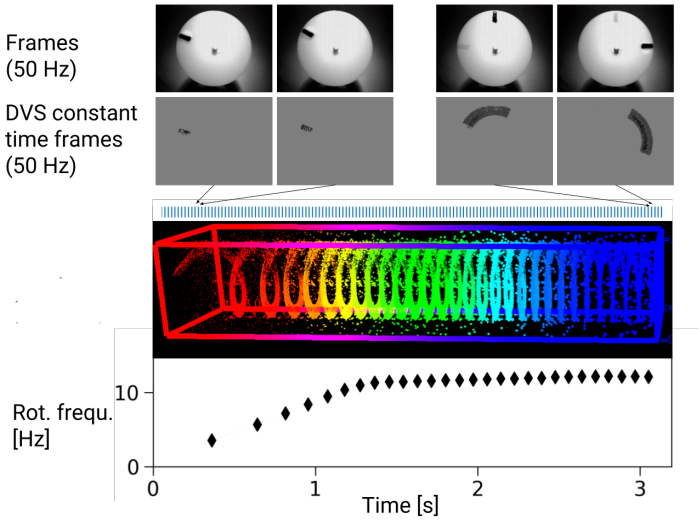


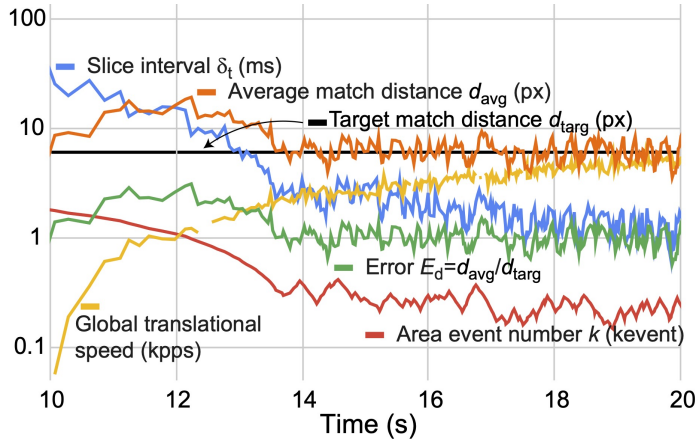
FIGURE 6.18: Rotating dot input (adapted from [221]).

accuracy of flow. We measured the GT speed of the dot using a tracker. Using a fixed $k = 600$ is OK for low speed, but when the dot moves quickly, the flow results are much less accurate than when k adapts d_{avg} towards d_{targ}. In summary, when the input scene dynamics varies over time, the feedback control of k can stabilize the BMOF vectors in the middle of their dynamic range.

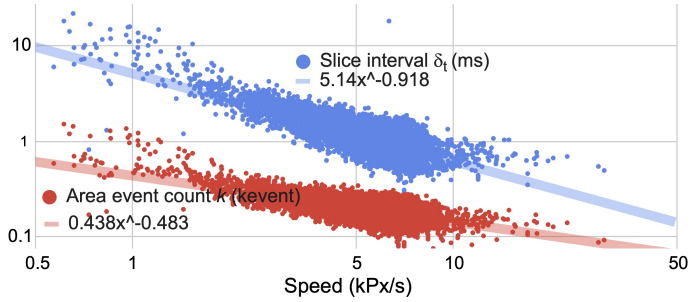
Table 6.4 lists the processing time (computational latency) of ABMOF and its variants compared with [189] [2]. The dense ABMOF can process dense OF using only 1 us/event, or at rate of 1 MHz. It is about the same speed as [189]. The sparser ABMOF+EFAST and ABMOF+SFAST are more than 10X faster and can process event rates up to 16.6 MHz.

Xilinx Vivado reports that the EDFLOW IP blocks consume about 1W (Fig. 6.20). EDFLOW is thus about 75X times more energy efficient for

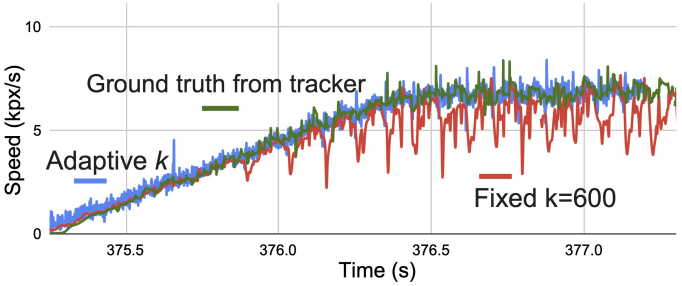
2 The processing time per event for *Ev-Flownet* is not a valid metric because it uses a constant FPS event volume input representation. Its processing cost is constant; the time per frame is 40 ms using the smaller *Ev-Flownet* network variant on an NVIDIA 1050 GPU with a power consumption of 75W. For a CNN that uses "constant count" event voxel grid input [11](Sec. 3.1, Fig. 3), then we could assume *e.g.*, 10k events per input and compute that the time per event would be 40ms/10 kev=4 us/ev.



(a) ABMOF area event number k feedback control *vs.* time



(b) Rotating dot



(c) ABMOF area event number k feedback control *vs.* time

FIGURE 6.19: ABMOF feedforward and feedback control of slice event count. (**a**) Rotating dot input (adapted from [221]). (**b**) Plots of ABMOF quantities over time. (**c**) Slice interval and area event number versus dot speed. (**d**) Comparison of accuracy of dot speed vs time with and without adaptive k.

ABMOF and more than 450X more energy efficient for ABMOF+SFAST than the CNN.

ABMOF (full)	1
ABMOF+EFAST	0.06 - 0.1
ABMOF+SFAST	0.06 - 0.1
Aung [189]	0.72-1.19

TABLE 6.4: Computational latency comparison. Time per DVS event in microseconds.

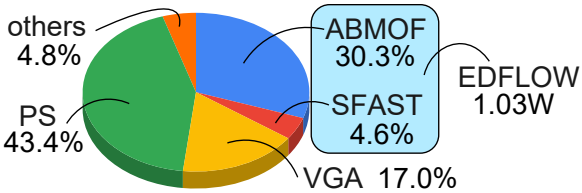


FIGURE 6.20: The composition of dynamic power consumption in the Zynq XC7Z100 SoC. Total power is 2.96W and the EDFLOW IP burns 1.03W. Wall plug power measured from power strip is 10.1W and includes the power pucks, the FPGA power board, USB, DAVIS346, VGA driver, DRAM, and other PCB components. Total power varies about 400mW depending on activity.

Table 6.5 summarizes EDFLOW FPGA resources. EDFLOW uses about the same logic as [193], and more logic and memory than [189], but only a fraction of the whole FPGA. We see that the largest usage is of BRAMs (25%) and DSPs (33%). The adders in the DSPs are used for the SAD computations.

6.6 SUMMARY AND DISCUSSION

In this chapter, we use SFAST as the keypoint detector for ABMOF. By calculating OF only on "corner events", ABMOF solves the aperture problem better and thus improves OF accuracy, with the tradoff of lower OF density. We compared two corner detection methods EFAST and SFAST. The results show that SFAST achieves more accurate OF than EFAST. The

Module	LUTs	FFs	BRAM_18Ks	DSP48Es
AER	2147 (1%)	4845	10 (1%)	0
SFAST	5806 (2%)	3987	32 (2%)	0
ABMOF	45501 (16%)	25323	348 (23%)	669 (33%)
Total	53450 (19%)	34160	390 (26%)	669 (33%)
Aung *et al.* [189]	15180	30350	138	16
Seyid *et al.* [193]	36430	42500	NA	48

TABLE 6.5: XC7Z100 FPGA resources for main IPs.

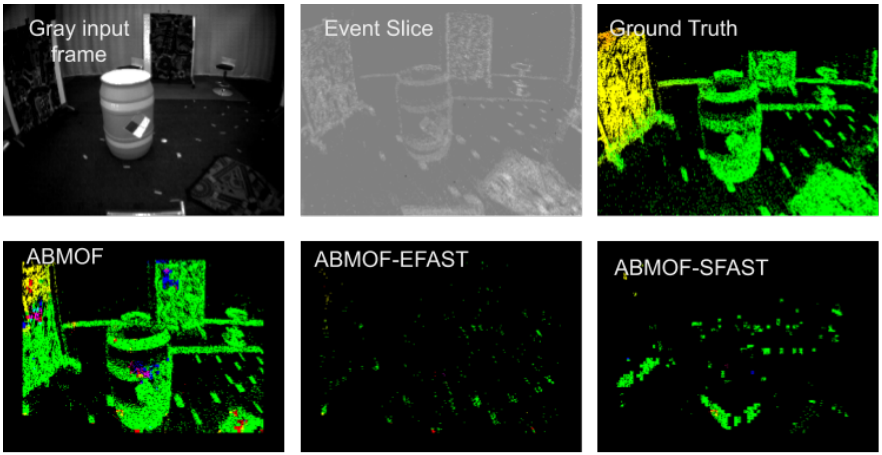


FIGURE 6.21: Optical flow computed by adaptive block matching optical flow (ABMOF) and its variants. Top row from left to right is the gray input frame captured by DAVIS's APS part, event slice with events accumulated for 50ms and the optical flow ground truth. Bottom row from left to right is the pure ABMOF result, ABMOF estimation result only on timestamp image EFAST corners and ABMOF estimation result only on accumulated event count SFAST corners. Tested on indoor_flying_1 from [121]. Best viewed in color.

use of keypoints is optional, and EDFLOW includes the semidense ABMOF mode where all events that can be processed without overflowing

the pipeline are processed, and with OF vectors labeled as keypoint OF events.

Comparing the ABMOF variants with CNN-based OF, ABMOF is hundreds of times quicker and more power efficient, but is also less accurate. Compared with previous DVS OF methods, ABMOF uses a complex multiscale CTF BMOF method that matches complex patterns; however compared to state of the art video compression methods like HEVC, ABMOF is simple [222]. Our work does not exploit the gray-scale frame APS output available from the DAVIS [8, 177]. Future work could fuse its DVS and APS output.

Although it can be argued that EDFLOW does not make use of the precise DVS event timing, Fig. 10 from [5] shows that DVS event timing jitter can exceed 1 ms, which makes methods like LP work poorly. However, EDFLOW retains the activity-driven computation initiated by the DVS brightness change detection. All the OF computations done by EDFLOW are initiated by events, and EDFLOW's event slice rotation is determined by the highest local event rate. Although the global event slices have the drawback that they can be too sparse in low-activity regions, their synchronous timing makes the logic design much simpler, and they are free of motion artifacts like those seen in rolling shutter imagers. EDFLOW's activity-driven computing is the main ingredient of efficient brain computing that is increasingly being exploited by sparsity-aware DNN accelerators [223–227].

All modules were implemented on the `Davis346Zynq` prototype camera. Accounting for the operations only in ABMOF, EDFLOW achieves 123 GOp/s at a clock frequency of only 100MHz, *i.e.*, the equivalent of 1230 Op/clock cycle[3]. This massive parallelism of the SAD computations is obtained by pipelining and block memory organization.

EDFLOW uses no multipliers and mostly 4-bit arithmetic. The FPGA implementation of ABMOF+SFAST uses a lot of BRAM (855 KB) and many DSPs (669). The 48-bit DSPs very inefficiently implement the SAD accumulators. The actual total SRAM usage is less than 300 KB[4]. As an ASIC, EDFLOW could be tightly coupled to the DVS and use a higher clock frequency. For a larger DVS, *e.g.*, 1 MPixel, ABMOF would require a significant 3.2 MB of memory, so it would be worth understanding if off-chip DRAM IO could be scheduled to use the required burst IO with event-

3 One SAD computation requires 3 math operations: subtract, absolute and accumulate. The total number of SADs to be computed is $(25 \times 25 + 13 \times 13 + 7 \times 7) * 49 = 41307$ SAD. Thus, the total operations are $41307 \times 3 = 123921$ Op. All of them are finished within 100 cycles.

4 Memory usage of ABMOF for the 346 *times*260-pixel DAVIS346 is dominated by slice memory: 346×260 pixels $\times (1 + 1/4 + 1/16)$ scales $\times$ 4 slices $\times$ 4 bits/pixel $= 1.9$ Mb $= 283$ kB.

driven BMOF. In ASIC, the power consumption would be reduced by more than 10X based on our experience [224]. EDFLOW would be straightforward to implement as an ASIC IP block, where the logic and memory blocks could be more tightly integrated and the SRAM could be optimized. The main difficulty is using HLS for the ASIC since HLS is not popular for ASIC design (Sec. 7.2.6).

The main limitation of a hand-crafted method like ABMOF is that it cannot model real world data like DNN methods. The large block size that EDFLOW uses also has difficulty detecting motion borders, which are informative for object versus background segmentation. Additionally, a local method like EDFLOW invariably has a larger number of outliers. However, it may not be necessary to use a large DNN for accurate end-to-end OF since OF is an intermediate result. It usually drives further processing such as VOD or SLAM. The inclusion of EDFLOW in the camera could enable a complete VOD pipeline, or it could simply offload low-level keypoint detection and OF to the camera. The OF could then serve as informative elaborated features, including further DNN processing. One could envision a future event camera that integrates the sensor, EDFLOW, and a sparsity-aware DNN accelerator. The size of the DNN could be reduced by using the informative low level computation done by EDFLOW.

This chapter concludes the technical content of the thesis. The next chapter summarizes the thesis work and concludes with an outlook for future work.

7

CONCLUSION AND OUTLOOK

"The brain is imagination, and that was exciting to me; I wanted to build a chip that could imagine something."

— Misha Mahowald, 1986

7.1 CONCLUSION

In this thesis, we introduced, to the author's knowledge, the first event-based hardware OF system that use corner detectors. The algorithms consist of a block-matching-based method ABMOF and a corner detector SFAST. Additionally, we also designed a new powerful general hardware platform that has the capacity to be extended to an event-based navigation or SLAM system for drones or robotics in the future.

ABMOF (see Chapter 3) extends from BMOF (see Chapter 2). BMOF is inspired from a network media processor LSI Logic DoMiNo™ . LSI Logic and other companies used BMOF in mass production video compression architectures since the 1990s [228, 229]. A local gradient measurement such as that used by LK is relatively cheap to compute, but it cannot account for nonlocal structures. For better OF estimation, video compression algorithms use BM in BMOF to match blocks of pixels between frames using a hierarchical CTF search. BMOF is not popular for software OF because it requires many operations that must be computed serially on a CPU. The key requirement for BMOF with DVS events is to collect DVS frames with high quality features.

ABMOF algorithm was introduced to adaptively vary the DVS event slice exposure for good block matching quality. The event slices are adaptively rotated based on the input events and OF results. The area event number slice rotation method, which adapts to scenes with varying levels of spatial sparsity, is used to accumulate events in rotating event slices. The slice rotation event count number is feedback controlled by the average optical flow matching distance. Compared with other methods such as gradient-based OF, ABMOF can efficiently be implemented in compact logic circuits. Results show that ABMOF achieves comparable accuracy to conventional standards such as LK. The main contributions of ABMOF are

new adaptive time-slice rotation methods that ensure the generated slices have sufficient features for matching, including a feedback mechanism that controls the generated slices to have an average slice displacement within the block search range. The CTF strategy makes it more robust to the large displacement motion, and the multiscale block match size is 25×25 pixels, and the flow vectors span up to 30-pixel match distance. The ABMOF shows a good accuracy result on various natural data scenes, including sparse and dense texture, high dynamic range, and fast motion exceeding 30,000 pixels per second.

Computing every event is not an efficient way. Additionally, some edge events would result in the *aperture problem*. Therefore, we implemented previously reported corner detector algorithm called EFAST [18] on MiniZed (see Chapter 4). EFAST originates from FAST. SFAST detects the corners by checking if a continuous pixel streak on the circle is centered on the event. Corners are detected by streaks of accumulated events on event slice rings of radius 3 and 4 pixels. It does not have any derivate operations and thus only adders and comparators are required. The maximum processing event rate is 10M eps. However, since it operates on a 32-bit TI, it requires a lot of memory for storage. It is also very compatible with ABMOF. SFAST is thus proposed.

SFAST (See Sec.6.4.2) is improved from EFAST. Similar to EFAST, it also tries to find the continuous streaks on the circles centered on the event. But it has several differences. First, it checks on the event count slices rather than TI. Second, it uses smaller circles whose radius are 1 and 2 pixels respectively. Third, it has a more strict hyper parameter to control the difference between the minimum of streak and the maximum of the non-streak. Fourth, it has an extra condition to check the streak shape. Both SFAST and ABMOF use event count slices, but SFAST only operates on the coarse-scale and thus saves a lot of memory. Comparing SFAST with EFAST, that improves the accuracy while saving at least 8X memory.

To implement and verify ABMOF and SFAST on hardware, a new powerful event-based camera platform `DAVIS346Zynq` (See Chapter 5) is designed. It has 512MB memory and 1G NAND flash. SD card is also supported. It supports two play modes: live mode and playback mode. It can read files from the SD card directly and then sequence them into the real live event stream for playback mode. It has a customized USB 2.0 high-speed and VGA interface. The DAVIS controller is also integrated on FPGA.

Combining hardware ABMOF and SFAST, the EDFLOW (See Chapter 6) is build. Flow vectors are only estimated at the corners. The EDFLOW

camera processes events at rates up to 16.6MHz for corner detection. At the corners, flow vectors are computed in 1 us at 100 MHz clock frequency. EDFLOW processes the sum-of-absolute distance block matching at 123 GOp/s, the equivalent of 1230 Op/clock cycle. EDFLOW is eight times more accurate than the previous best DVS FPGA optical flow implementation. Although it is less accurate than CNN-based optical flow, it burns about 100 times less power and is ten times quicker. EDFLOW also provides an easy way to change some parameters such as SFAST threshold and area event number threshold and output the real-time status such as corner event number statistic for the hardware IPs on the board. Other hardware configurations can also be changed very conveniently, such as skip corner detection, live mode selection, etc. The configuration process is done with a UART interface (see Appendix A.5).

7.2 OUTLOOK

Event-based camera has been in development for the last decade. Compared with the traditional frame-based camera's history, it is still a very new thing. However, in recent years, it has attracted increasing attention. There are still many problems required to be solved by the event-based vision community. This section shows some opinions and outlook of the author of this thesis work and the event-based community.

7.2.1 *Accuracy improvement*

When computer science researchers design an algorithm such as the optical flow algorithm for a computer vision problem, the accuracy usually has the highest priority. However, in the robotics community, real-time is also a critical factor to be considered. Like one coin has two sides, accuracy and speed are the two sides of an algorithm. Moreover, it is always a trade-off between them. Higher accuracy usually means heavier computation. EvFlownet achieves very high accuracy in the event-based OF area, but it is based on a desktop GPU solution. Compared with EvFlownet, the algorithms of EDFLOW, especially ABMOF, still needs to be improved in the future on the aspect of accuracy. Based on the real-time principle, several interesting directions could be tried:

- As shown in [193], current smooth constraints or regularisation items usually require derivative and floating-point computation; one possible direction is to explore some more hardware-friendly constraints.

- OF results still have many large outliers. Some spatial-temporal correlation filters or even RANdom SAmple Consensus (**RANSAC**) [154] could be tried in the future.
- The tracking process for the frame-based camera usually consists of three steps: feature detection, feature description, and matching. SFAST is only able to do the first step. Additionally, Brox and Malik [70, 230] show that an extra feature description stage as the refinement could improve the optical flow performance a lot. Therefore, combining with some event-based feature descriptors [231] with EDFLOW would also be an exciting direction.

7.2.2 *Optical flow as features for DNN accelerators*

When talking about computer science, a topic that cannot be avoided is deep learning. The wave of deep learning progress has impacted almost all subjects. How to improve this work in a deep learning direction? The most straightforward answer would be to design an event-based OF hardware accelerator based on DNN. However, from the viewpoint of the author, it might not be so good. OF is always used as a low-level processing method in the CV community. Although many works solve this problem with the popular deep learning method, the author does not think a deep-learning-based OF has too many applications. The reason is that OF often is not used as an independent application like VOD, SLAM, or action recognition. Most of the time, it servers as a preprocessing module in the system. In this case, a light-resource-consuming with moderate accuracy method would be a better solution.

However, it is still possible to combine it with deep learning methods. As shown in Figure 7.1, it uses OF as a low-level feature, and then the output features are fed into some hardware accelerators. In the robotics area, researchers have explored this direction, such as [232, 233]. In their work, they use OF results to estimate the displacement and rotation between two frames. The OF result image is treated as a color-coded image using the standard mapping of flow image to color and then fed into a CNN to extract the visual odometry information. Compared with training from scratch [234], OF provides more additional information and thus improves the accuracy of the system a lot.

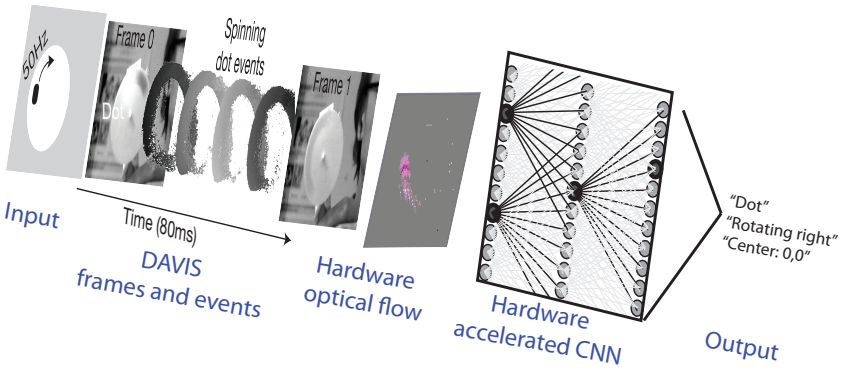


FIGURE 7.1: Use OF as a feature for other hardware accelerators.

7.2.3 *Combine with other sensors for sensor fusion*

The camera is a bearing sensor, which means it is not very accurate for distance measurement, and it is possible to recover the scale for monocular vision. To improve the accuracy and recover the scale, the event-based camera could be integrated with other sensors. Based on the current DAVIS camera including an IMU onboard, one quick possible direction is fusing the camera and the IMU. For example, EDFLOW only measures the flow on the 2D image, but with the help of IMU acceleration output, the depth (the missing scale) could be recovered by geometry methods. It could then be used in 3D reconstruction, VOD, or SLAM. [235] is a frame-based work for VOD based on OF and IMU. [29] explored tracking using OF and IMU-based on event-based cameras. [236] is another work that converts event-based OF to VOD. Nevertheless, there are still relatively few works for event-based cameras fusing 2D OF with IMU to obtain scene flow or other applications. In this area, two possible directions could be explored further. The first one is to improve the algorithm further, and the second one is to extend the EDFLOW. Compare and select from some existing hardware-friendly Visual-Inertial Odometry (VIO)algorithms based on OF and IMU, and then implement them on `DAVIS346Zynq`. This could make the low latency and low power consumption but fully self-navigation system possible for the event-based camera.

Final Clean

	EPE all	EPE matched	EPE unmatched	d0-10	d10-60	d60-140	s0-10	s10-40	s40+	
GroundTruth [1]	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	Visualize Results
GMA [2]	2.470	1.241	12.501	2.863	1.057	0.653	0.566	1.817	13.492	Visualize Results
SeparableFlow [3]	2.667	1.275	14.013	2.937	1.056	0.620	0.580	1.738	15.269	Visualize Results
MixSup [4]	2.685	1.213	14.685	2.863	1.034	0.542	0.571	1.671	15.628	Visualize Results
RAFT+NCUP [5]	2.692	1.323	13.854	3.139	1.086	0.636	0.635	1.844	14.949	Visualize Results
MFR [6]	2.801	1.380	14.385	3.075	1.112	0.772	0.674	1.829	15.703	Visualize Results
RAFTwarm+AOIR [7]	2.813	1.371	14.565	3.088	1.099	0.727	0.603	1.781	16.271	Visualize Results
RAFTwarm+OBS [8]	2.826	1.356	14.809	3.134	1.116	0.735	0.631	1.832	16.117	Visualize Results
RAFTv2-OER-warm-start [9]	2.831	1.396	14.536	3.109	1.133	0.742	0.628	1.798	16.259	Visualize Results

FIGURE 7.2: The screenshot of MPI-Sintel benchmark website at the time of writing.

7.2.4 *Event-based optical flow benchmark*

The frame-based optical flow community has established a complete set of benchmarks for optical flow estimation. The three most popular among them are Middlebury [90], MPI-sintel [91] and KITTI [93]. Researchers could publish their results on these three datasets and show the rankings and nice visualization results on the websites. Fig. 7.2 is the latest screenshot of the MPI-Sintel benchmark dataset. The table on the website clearly shows the accuracy ranking, the metrics used, and a button for visualizing results. It helps the researchers know the state-of-the-art on this dataset quickly and clearly. However, for the event-based optical flow community, such kind of benchmark has not existed yet. To date, MVSEC [24] is probably the largest event-based optical flow dataset providing ground truth. However, this dataset is not targeting solely optical flow. Additionally, there is not a standard metric as their frame-based counterparts. It is not easy for people to compete on the same dataset and know the ranking quickly. Creating a pure event-based optical flow dataset and proposing some standard metrics to publish their results quickly and show their rankings could be a big step to promote this area further. Since making an

event-based optical flow dataset with labeled ground truth is difficult, one possible solution might use state-of-the-art simulation tools such as [23] to convert the frame-based popular datasets to event-based datasets. This could help establish the benchmark more quickly, and provide a way to compare with the frame-based methods.

7.2.5 *Event representation*

Event representation is also a very interesting question to be explored. Unlike the frame-based camera, the data is formed by a fixed size frame and generated in a fixed sample one by one. The output of the event camera is only a stream, and one event usually contains little information. Therefore, organizing these events to form a meaningful representation is always a very core question for event-based data processing. In ABMOF, we use the event count slice as the representation. However, to make it more robust, we use a method called area event number and a feedback mechanism to dynamically adjust the duration of an event slice. EFAST uses the TI as the event representation. There are many other event representation methods used in the research. Such as in [237], they use a four-dimension tensor for the event representation. In [237] they also compared some common representation methods in the community and show how different representation affects the performance of the DNN. It finds that the event spike tensor with time stamp measurements has the highest accuracy on the test set for both N-Cars [238] and N-Caltech101 [239] for object recognition task [237]. Refer to [237] to check more details about different event representations. Therefore, in the future, exploring some different event representation methods for ABMOF and SFAST would also be an interesting direction.

7.2.6 *ASIC silicon area/power estimates*

To make EDFLOW more compact and lower power consumption, ASIC would be a step further, but ASIC affordability comes with volume. Custom architectures are usually part of the niche market and can be ill-fitting for ASIC design. Here, we make a brief estimation about the cost for this. We assumed that an upcoming stacked, back-illuminated vision sensor technology would be used for an advanced DAVIS and that it would use a 28nm digital process and attempted to estimate the silicon area and power for implementing EDFLOW. The main difficulty has proved to be the HLS

memories; the output of HLS conversion is Verilog HDL, but it includes many uses of particular BRAM macros. The other difficulty is the timing issue. When HLS generates the RTL circuit from the C++ code, it can handle the timing problem automatically. It can generate the circuit under the user's satisfaction without too much user intervention. However, this problem requires to be considered very carefully in the ASIC design, especially when there is a BRAM macro. Our development is currently partially complete, but we do not have easy access to an SRAM memory compiler to overcome this hurdle.

7.2.7 *Is the event camera at the dawn of a new computer vision era?*

When thinking about commercial applications for event-based cameras, one important factor cannot be ignored is the price. Until now, these sensors from all manufacturing companies are still sold at a very high price. Some of them are even comparable to a super-high-speed frame-based camera. This prevents a wide usage in the industry and, on the other hand, increases the entrance threshold for researchers. This forms a bad loop: the high price (caused mainly by small scale production) makes few researchers in the community, which results in few applications, and makes it more difficult to be accepted by the market and thus keeps the price high. Some good news is that in recent years, increasing event-based datasets are released, and research could be done even without a real event-based camera on hand. On the other hand, from [1, 22] to the most recent [23], the model of DVS simulators has also become increasingly realistic.

This scene makes us think about a similar historical situation. At the dawn of motion pictures, Edward Muybridge was interested in photography as a tool for capturing data. It only secondarily was taken up by others as a sensational entertainment medium. However, it has always advanced the purposes of science, and it will continue to do so until the last producer has choked on his last cigar. This is the second revolution. The second revolution occurred when CMOS technology was first introduced in the image sensor. The CMOS camera has bigger noise, low resolution and is more expensive than the CCD camera during the early time. However, the CMOS camera is the mainstream technology in the image sensor area today. Now event-based camera comes to a similar threshold point, will it be the dawn of the event-based camera's bright future?

To sum up, the event cameras DVS and DAVIS provide high temporal resolution, sparse, low-latency data. It can work in a higher-dynamic-

range environment and consume several mWs. As discussed before, it is a potential sensor for high-speed applications, such as drones or robotics. Nevertheless, there are still some limitations to these cameras. The major obstacle for DVS or DAVIS is that the resolution is still low compared to the mainstream frame-based camera. Currently, the largest spatial resolution published on the event camera is about 1 Mpixel (1280x960) [170]. This is almost nothing even compared to a phone's camera.

However, the author believes that the price could drop precipitously once this technology enters mass production. Moreover, it attracts increasing interest from investors and giant players in recent years, such as the acquisition by OmniVision Technologies of Celepixel and the acquisition by Sony Semiconductors of insightness. From the author's view, event cameras would follow a similar success as their counterpart, frame-based cameras, though it will require effort in time and energy of talented people worldwide.

A

APPENDIX

A.1 BUILDING A MINIZED SDSOC PLATFORM

As we described in Chapter 4, we should prepare a SDSoC platform before we use SDx. This platform consists of two parts. One is the software part. This part provides the bootloader, Linux image, and libraries. The software part is built by a tool called Petalinux, which is also from Xilinx. The other is the hardware part. This part is described by a Device Support Archive (**DSA**) file. This file is generated from a basic hardware design using Vivado.

This section shows how to build an SDx platform for the MiniZed board. The source files for building this platform are released on GitHub. This repo is called mz_petalinux_SDx. The versions of Vivado tools used in this thesis are 2018.1.

Check https://github.com/wzygzlm/mz_petalinux_SDx to obtain all building files.

A.1.1 *Hardware*

There are three folders “board_vivado", “minized_petalinux_prjs" and “sdx" under the root folder of the repository.

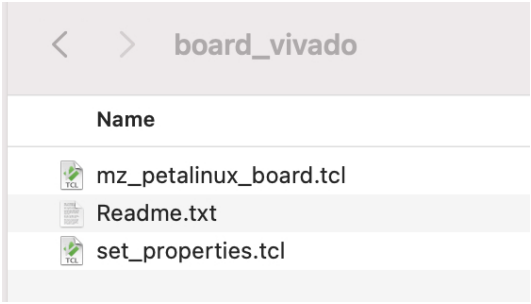


FIGURE A.1: The content of “board_vivado" folder.

Fig. A.1 shows the files in the folder “Board_vivado". It mainly contains two tcl files: mz_petalinux_board.tcl and set_properties_.tcl. The sdx folder contains the complete pre-built-in platform files.

Mz_petalinux_board.tcl is used to generate the basic hardware Vivado projects. Source mz_petalinux_board.tcl can generate a predefined Vivado project for MiniZed. Alternatively, users can also create their own hardware projects from scratch. After synthesis, implementation, and generating the bitstream, source the second file “set_properties.tcl" to describe what resources this platform provides. The properties include platform name, clock, AXI ports, interrupts, etc. The last step is to generate the (**DSA**) file. It archives the board tcl files, initial hardware scripts, IP files used in the design, etc. SDSoC could reconstruct the hardware design based on this DSA file.

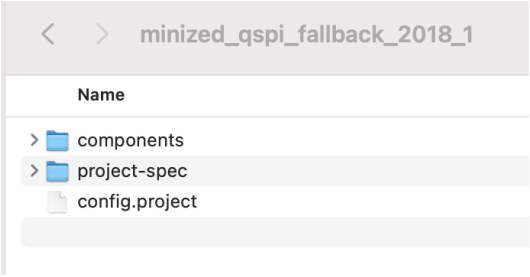


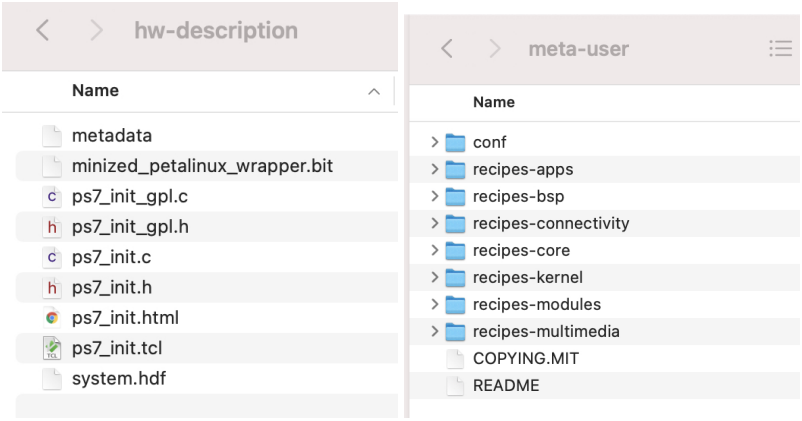
FIGURE A.2: The folder structure of a typical petalinux project.

A.1.2 *Software*

A typical petalinux project contains two folders: components and project-proc. Components contain some built-in headers and libraries from Xilinx. The Project-spec folder stores the customized configuration by users. It has three subfolders: *configs*, *hw-description*, and *project-spec*.

The *configs* folder contains two subfolders: *config* and *rootfs_config*. The first subfolder is for system configuration. *Rootfs_config* stores the configuration for Rootfs. Rootfs is a file system based on RAM.

The *hw-description* folder has the files for the hardware. Most of them can be extracted from the DSA file we mentioned above. One file called Hardware Description File (.hdf) could be exported from Vivado.



(a) Hw-description folder. (b) Meta-user folder.

FIGURE A.3: Hw-description and meta-user folder.

Meta-user folder has several subfolders: *conf*, *recipes-apps*. *recipes-BSP*, *recipes-connectivity*, *recipes-BSP*, *recipes-core*, *recipes-kernel*, *recipes-modules*, and *recipes-multimedia*. Petalinux uses Yocto as the builder. Conf folder stores the configuration for the builder. Yocto adopts layers to manage the packages to be built. Every package is stored in a format of .bb file. All other folders with a prefix “recipes" contain customized packages. *E.g.*, *recipes-kernel* has user’s modifications or patches to the kernel. *Recipes-core* stores all applications added by the user. Remember to update *petalinux-image.bbappend* in this folder after modifying *recipes-app*.

After configuration, the next step is to build. The command is `petalinux-build`. It takes a while ranging from one hour to several hours. Make sure to prepare enough space for the intermediate artifacts. It can consume up to even more than 100GB. Details about petalinux tools are in [240].

If everything goes well, the result of petalinux includes a Linux kernel image file, bootloader, and rootfs file. Copy all these files to the software folder of a SDSoC platform. In conjunction with the DSA file from the hardware part, a complete SDSoC platform is finished constructing.

A.2 SOME TRICKS OF HLS OPTIMIZATION

Although Vivado HLS uses C++ as input, there are some techniques or tricks that might be useful while writing the code.

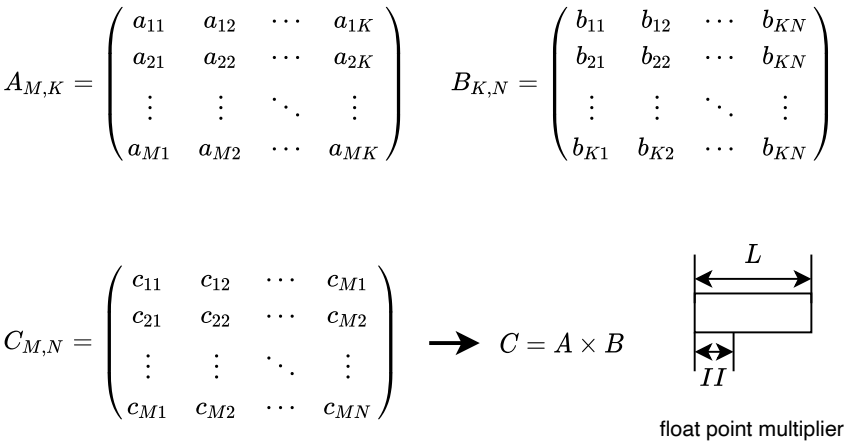


FIGURE A.4: The float matrix multiplication problem. There is a float point multiplier on the hardware with an initial interval II and Latency L.

A.2.1 *Interleaving technique*

The data interleaving technique is a very popular technique used in hardware architecture. It is primarily used to solve loop-carried dependency. Loop-carried dependency means the input of a computation loop depends on the result of its previous computation loop. This makes the next loop cannot be started until the previous loop is finished. To solve the loop-carried dependencies, we need to make the interval between these two loops have enough time. Here we use the float matrix multiplier as an example to show how to optimize the code using interleaving.

Fig. A.4 shows the float matrix multiplication problem. The timing performance of the basic hardware unit float point multiplier is also shown in Fig. A.4.

The first solution we used is writing the code as a typical software design as shown in Listing A.1 which is we first computing $a_{11} \times b_{11}$ and then

$a_{12} \times b_{21}$ and so on. Its computing order and the timing block are shown in Fig. A.5(a). Due to the loop-carrying dependency (line 11 in Listing A.1), the initial interval of the design could only achieve L instead of II.

Code is rewritten in Listing A.2. The difference between this code and the original code is that the k loop and m loop are switched. Therefore, the computing order is changed. $a_{11} \times b_{11}$ is followed by $a_{21} \times b_{12}$ instead of $a_{12} \times b_{21}$. Fig. A.5 (b) shows the computing order and timing block after optimization. This design could achieve the initial interval as the same value as the basic float point multiplier. The reason is by changing the computing order, $a_{11} \times b_{11}$ and $a_{12} \times b_{21}$ are reordered in different loops and their interval is changed to $(M-1) * II + L$ which is far more than L. The loop-carried dependency is thus solved.

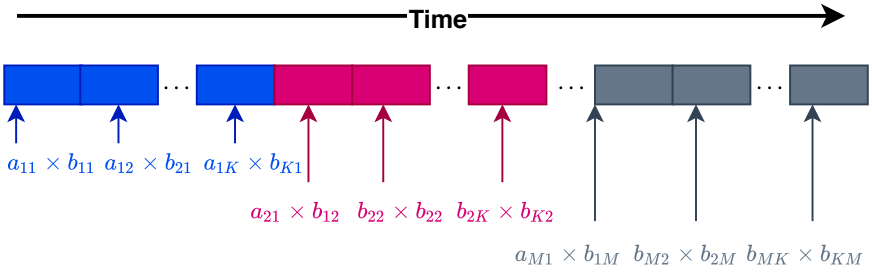
```
1  void FloatMatrixMul(float A[N][K], float B[K][M], float C[N][M])
2  {
3    for(int n = 0; n < N; n++)
4    {
5      for(int m = 0; m < M; m++)
6      {
7        float acc = 0;
8        for(int k = 0; k < K; k++)
9        {
10 #pragma HLS PIPELINE
11         acc += A[n][k] * B[k][m];
12       }
13       C[n][m] = acc;
14     }
15   }
16 }
```

LISTING A.1: Float matrix calculation in the normal order.

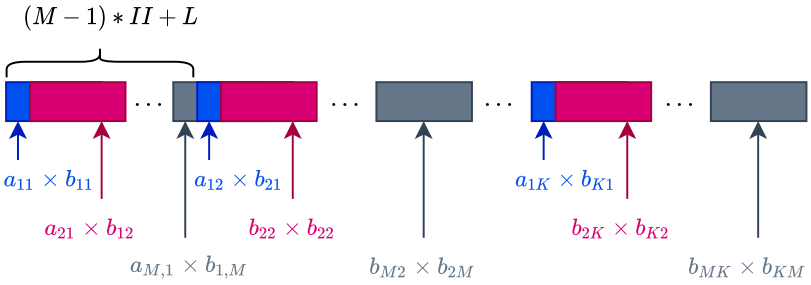
```
1  void FloatMatrixMulInterleaving(float A[N][K], float B[K][M], float C[N][M])
2  {
3    for(int n = 0; n < N; n++)
4    {
5      float acc[M];
6      for(int k = 0; k < K; k++)
7      {
8        float a = A[n][k];
9        for(int m = 0; m < M; m++)
10       {
11 #pragma HLS PIPELINE
12         float prev = (k == 0) ? 0 : acc[m];
13         acc[m] = prev + a * B[k][m];
14       }
15       for(int m = 0; m < M; m++)
```

```
      {
        C[n][m] = acc[m];
      }
    }
  }
}
```

LISTING A.2: Float matrix calculation in an interleaving order.



(a). The timing of Listing A.1. Compute in a normal order



(b). The timing of Listing A.2. Compute in an interleaving order

FIGURE A.5: The timing diagram of the float matrix multiplication code before and after using interleaving.

This is a quite useful trick when encountering the loop-carried dependency. By transposing the inner loop and outer loop, The two items that are neighbors within the same loop could be reordered into different loops and thus improve the throughput.

A.2.2 *Apply dataflow to several simple PE*

While designing the core part of hardware ABMOF, we divided the SAD into three small PEs: ColSAD, RowSummer and FindStreamMin (See Chapter 6). This section gives an elaborated explanation of this technique.

This technique firstly partitions a complex function into several simple PEs and then arranges them in a dataflow architecture. Dataflow [160] is a very powerful tool in Vivado HLS. Similar to the pipeline, this directive is also used to decrease II to increase the system's throughput. The difference between them is that dataflow has a higher level than pipeline. It is always used at the function level. The *dataflow* directive is used on the top-level function, so all submodules are connected one by one via FIFO, making the whole system more efficient. Dataflow makes every submodule are established with a consumer-producer model. It guarantees that the consumer module cannot start until the producer's output is ready by using ping-pong RAM or FIFO to synchronize among modules. Every process circuit could work in parallel with this mechanism, and the consumer unit will not get fake data. The dataflow makes the whole system work like a streamlined factory in which every process unit produces and consumes data simultaneously. This consumer-producer model results in the II of the whole system are determined by the submodules' maximum instead of the sum of all submodules.

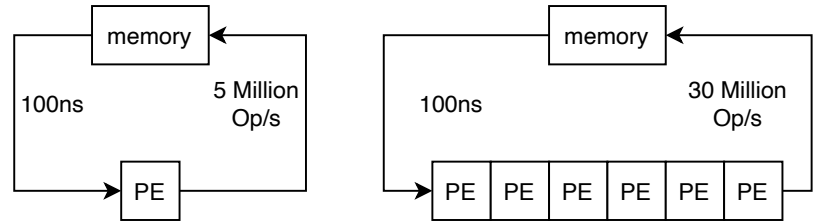


FIGURE A.6: Dataflow machnism. Adapte from [241].

Dataflow in HLS is similar to the concept of systolic system [241]. Differing from the systolic array that arranges many instances of the same PE units, dataflow can connect different PEs, which makes it more flexible. One of the most critical parts of the Tensorflow Processor Unit (**TPU**) is based on the systolic array [242].

Fig.A.6 illustrated the advantage of the systolic system intuitively. The delay of reading from and access to the memory is assumed 100ns. If only one PE is used as shown in Fig. A.6(a), the throughput is almost only determined by the delay between the PE and the memory. The maximum of the throughput of the system thus is 5 million Operations Per Second (OPS) at most. In Fig. A.6(b), six same PEs are connected, and the data is fed into all PEs in sequence as if a data stream flows into the PE array. This hardware structure is called the systolic system. The same data could be reused six times, which means the throughput of this system could be six times higher than the system in Fig. A.6(a). Hence 30M OPS could be achieved.

Back to the example that we mentioned at the beginning of this section. Let us take scale 0 as an example for calculation. The II for these three PEs are $\lceil b_f \div NPC \rceil \times (2 * r + 1)$, $\lceil b_f \div NPC \rceil \times (2 * r + 1)$ and $(2 * r + 1)$ respectively. For details of NPC, check Sec.6.4.4. Substitute the value from Table 3.1 with $b_f = 25$, $r = 3$ and $NPC = 2$, the real values are 91, 91 and 7. If we simply contacted these three PEs ColSAD, RowSummer, and FindStreamMin in one module. The II of this complicate module is about $91 + 91 + 7 = 189$. However, if we divided into three independent PEs as we did, the II is the maximum, which is around 91. It thus helps increase the throughput a lot.

To summarise, in contrast to conventional pipelining, PEs arranged in a dataflow architecture are scheduled separately when synthesized by the HLS tool. There are multiple benefits to this [162]:

- Different functionality runs at different schedules. For example, issuing memory requests, servicing memory requests, and receiving requested memory can all require different pipelines, state machines, and even clock rates.
- Smaller components are more modular, making them easier to reuse, debug, and verify.
- The effort required by the HLS tool to schedule code sections increases dramatically with the number of operations that need to be considered for the dependency and pipelining analysis. Scheduling logic in smaller chunks is thus beneficial for compilation time.
- Large fan-out/fan-in is challenging to route on real hardware (i.e., 1-to-N or N-to-1 connections for large N). This is mitigated by partitioning the components into smaller parts and adding more pipeline stages.

A.2.3 *Miscellaneous tips*

This section presents some more useful tips for HLS optimization. Some of them help increase the throughput, and some of them might be used to save resources. Hardware design is a balance art between speed and resources. Apply them according to your requirements. More optimization tips we recommend readers to refer to [162, 243].

Duplicate instances to increase Parallelism. HLS synthesize the C/C++ code as follows: Top-level function arguments synthesize RTL I/O ports. C/C++ functions synthesize into blocks in the RTL hierarchy. If the C/C++ code includes a hierarchy of modules or entities with one-to-one correspondence with the original C hierarchy, all instances of a function use the same RTL implementation or block. The user could also implement several instances of the same function to make the system run in parallel if the resource is enough by setting the correct directive. This idea is similar to the concept NPC in Sec.6.4.4.

Try different block-level protocols. Every IP generated by Vivado HLS will have a block-level protocol to be accessed/controlled by other circuits. The default protocol is the ap_hls protocol. It is vital to choose an approximate block-level protocol for the IP since the resources consumed will significantly differ when integrated into the system. After choosing the block-level protocol, the port-level protocol is also required to be determined. HLS will generate all C driver files for this IP to be easily used in SDx or Vivado circuit design.

Be careful to use *allocation* directive. Vivado HLS provide a directive called *allocation* to increase the reusability of some resources. Sometimes, allocation does not reduce the LUTs since sharing also increases the multiplex, and some registers will be stored in several stages and consume more LUTs. As a result, the number of LUTs saved by allocation might be compensated by the number increased by its side effect, and sometimes the side effect is more substantial.

Remove abundant bits. For example, if the counter of one loop only requires 8bits, then use the necessary bit width. If int type is used, then HLS will synthesis a register with 64bits width, and it will increase a lot of area in some cases.

Make the bid width of data to multiple of 18. This tip only applies to Xilinx's hardware. The reason is that most of the BRAMs units on Xilinx's FPAG are 18-bit width. Set the data bit width as the multiple of 18 could help increase the utilization of BRAMs.

Use HLS Stream as much as possible. Vivado HLS provides a data type called stream to mimic the stream data such as video streams and event streams in the natural world. The stream could be easily implemented as FIFOs on hardware. FIFO does not require an address. Small FIFOs can even be implemented with shift registers and do not consume BRAMs. Therefore, FIFO is a very cheap resource on hardware.

On-chip data exchange. The bottleneck of many algorithms implemented on FGPA is the data exchange between the PL and the PS. How to exchange data faster between PS and PL is also a very hot topic in neural network accelerator [244]. If the buffer or temp memory is not big, instantiating them on the FPGA would be better.

Avoid float computing and try to replace them with bit shift. One example is from designing the feedback mechanism on ABMOF. The control factor is a decimal number. For jAER, this value is 0.05. We approximate it to $1/16$ on hardware since this value does not have to be fixed to 0.05.

Use cycle shift read and write for reading narrow data from a wide data. The details and example of this tip check Sec.6.4.2.1.

A.3 VGA PROTOCOL AND TIMING DIAGRAM

The interface of VGA consists of 15 pins and is distributed in 3 rows with 5 pins per row. Among them, 3 pins are used for brightness and 2 for synchronization signals. It carries three-color analog components RGB (red, green, and blue) and two synchronization signals for controlling the transfer. The other 5 ground pins form a loop with the above signal pins. One synchronization is for horizontal synchronization, and the other is for vertical. Formerly, 4 pins carried Monitor ID bits and a +12V DC pin. VESA DDC redefined these 5 pins. 3 pins were used for the I^2C interface of the EEPROM. It includes a +5V DC pin for the EEPROM power supply, a I^2C data pin, and a I^2C clock pin. The remaining 2 pins are reserved. VGA is transferred in analog form. We used a VGA DAC board to convert the digital signals to analog signals. The 2 synchronization signal pins control the transfer order among the 5 signal pins, and they determine the resolution.

The timing diagram of these 2 synchronized signals is shown in Fig. A.7. In Fig. A.7, VS is the vertical synchronization signal, and HS is the horizontal synchronization signal. a is the horizontal synchronize pulse, b is the horizontal back porch, c is the horizontal active time, d is the horizontal front porch, e is the horizontal total time. o is the vertical synchronize pulse, p is the vertical back porch, q is the vertical active time, r is the ver-

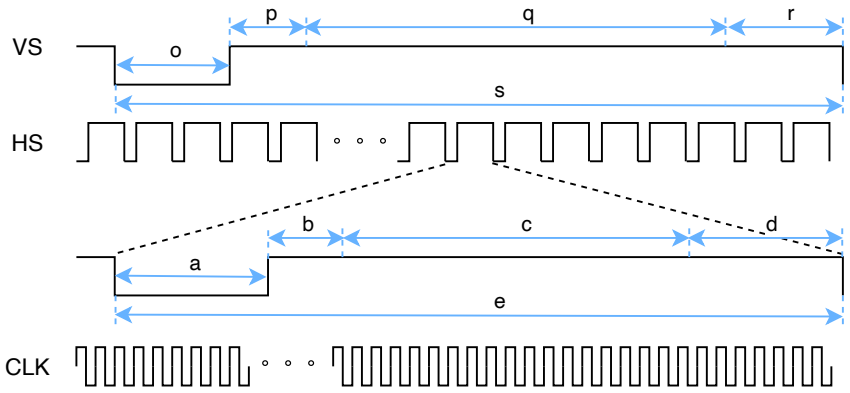


FIGURE A.7: VGA HS and VS timing

tical front porch, s is the vertical total time. The above values in different resolutions are shown in Table. A.1.

Display mode	Clock (MHz)	Horizontal timing (#clocks)					Vertical timing (#lines)				
		a	b	c	d	e	o	p	q	r	s
640x480@60	25.175	96	48	640	16	800	2	33	480	10	525
640x480@75	31.5	64	120	640	16	840	3	16	480	1	500
800x600@60	40.0	128	88	800	40	1056	4	23	600	1	628
800x600@75	49.5	80	160	800	16	1056	3	21	600	1	625
1024x768@60	65	136	160	1024	24	1344	6	29	768	3	806
1024x768@75	78.8	176	176	1024	16	1312	3	28	768	1	800
1280x1024@60	108.0	112	248	1280	48	1688	3	38	1024	1	1066
1280x800@60	83.46	136	200	1280	64	1680	3	24	800	1	828
1440x900@60	106.47	152	232	1440	80	1904	3	28	900	1	932

TABLE A.1: VGA refresh rate for various resolutions

A.4 USB 2.0 PROTOCOLS INTRODUCTION

USB uses a tiered-star topology. Every USB system has only one USB host. Host and all devices share the same bus. Every USB device on the bus has

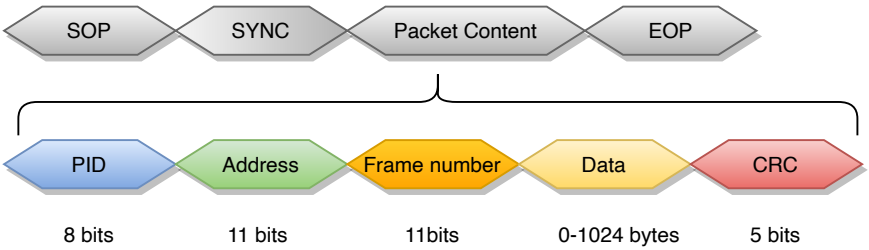


FIGURE A.8: The bit field of the USB packet.

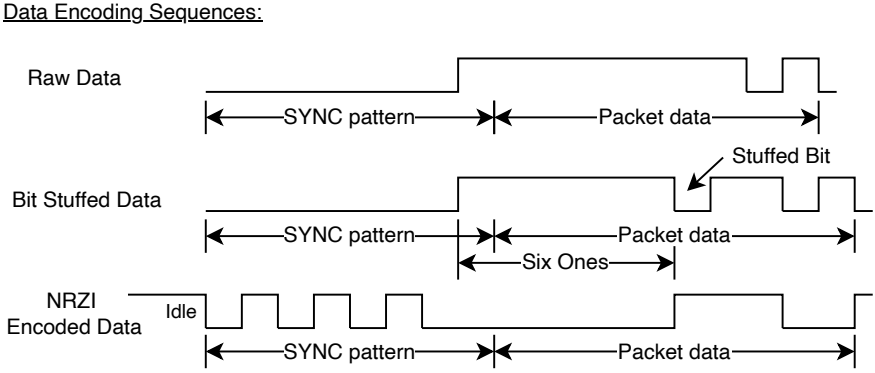


FIGURE A.9: Data encoding sequence with bit stuffing.

its own address. The maximum device number on a bus is 127. Within the USB device, there are many endpoints. The maximum of endpoints a USB device can have is 15. Endpoints can be categorized into control and data endpoints. Every USB device must provide at least one control endpoint at address 0 called the default endpoint or Endpoint0. This endpoint is bidirectional. That is, the host can send data to the endpoint and receive data from it within one transfer. The purpose of a control transfer is to enable the host to obtain device information, configure the device, or perform control operations that are unique to the device. Data endpoints are optional and used for transferring data. They are unidirectional, have a type (control, interrupt, bulk, isochronous) and other properties. All those properties are described in an endpoint descriptor.

Encoding of the data transmission on the bus follows the bit stuffing and NRZI. NRZI is a method of mapping a binary signal to a physical signal

PID Type	Value	Name	Description
Token	0001B	OUT	Address + endpoint number in host-to-device transcaction
	1001B	IN	Address + endpoint number in device-to-host transaction
	0101B	SOF	Start-of-Frame marker and frame number
	1101B	SETUP	Address + endpoint number in host-to-device transaction for SETUP to a control pipe
Data	0011B	DATA0	Data packet PID event
	1011B	DATA1	Data packet PID odd
	0111B	DATA2	Data packet PID high-speed, high bandwidth isochronous transaction in a microframe
	1111B	MDATA	Data packet PID high-speed for split and high bandwidth isochronous transactions
Handshake	0010B	ACK	Receiver accepts error-free data packet
	1010B	NAK	Receiving device cannot accept data or transmitting device cannot send data
	1110B	STALL	Endpoint is halted or a control pipe request is not supported
	0110B	NYET	No response yet from receiver
Special	1100B	PRE	(Token) Host-issued preamble. Enables downstream bus traffic to low-speed devices.
	1100B	ERR	(Handshake) Split Transaction Error Handshake (reuses PRE value)
	1000B	SPLIT	(Token) High-speed Split Transaction Token
	0100B	PING	(Token) High-speed flow control probe for a bulk/control endpoint
	0000B	Reserved	Reserved PID

TABLE A.2: USB packet types and its PID number and description. Adapted from [245].

for transmission over some transmission medium. It uses '0' to a signal transition and '1' for no change. To avoid the voltage on the transmission line staying constant for too long, bit stuffing is introduced to solve this problem. It requires that if there are six consecutive '1's occur on the bus, an additional '0' should be appended. For the decoding process, if detected six consecutive '1' and followed by a '0', the last '0' would be removed. An example of the USB data encoding is shown in A.9.

Signal	Line state	Description	Low speed (D- pull up)		Full speed (D+ pull up)	
			D+	D-	D+	D-
J	Same as idle line state	This is present during a transmission line transition. Alternatively, it is waiting for a new packet.	low	high	high	low
K	Inverse of J state	This is present during a transmission line transition.	high	low	low	high
SE0	Single-ended zero	Both D+ and D- is low. This may indicate an end of packet signal or a detached USB device.	low	low	low	low
SE1	Single-ended one	This is an illegal state and should never occur. This is seen as an error.	high	high	high	high

TABLE A.3: Some terminology used to represent some common USB line states. Adapted from [245].

Some terminology used for USB data transmission encoding is shown in Table A.3. The minimum data unit on the USB transmission unit is a packet. The bit field of a packet is shown in Figure A.8. It usually has 4 sections: Start of Packet (SOP), SYNC, packet content, and End of Packet (EOP). SOP means the start of a packet. A USB packet begins with an 8-bit synchronization sequence, 00000001B. This sequence is called the SYNC field. That is, after the initial idle state J, the data lines toggle KJKJKJKK. The final 1 bit (repeated K state) marks the end of the sync pattern and the beginning of the USB frame. The packet begins with a 32-bit synchroniza-

tion sequence for high-bandwidth USB, which is 32 '0's following by one '1'.

In the packet section, it usually has the following fields: Packet Identifier (PID), address, frame number, data, and Cyclic redundancy check (CRC). PID indicates the type of the packet. The meaning of PID field is shown in Table A.2. Address filed is an 11-bit field, and it consists of a 7-bit device address and a 4-bit endpoint address. The frame number is an 11-bit field to indicate the frame number of the packet. The data field holds the actual data to be transferred, and its size is variant according to the transfer type. The max size is 1026 bytes. CRC is used to make sure the packet is correct. Packet content's fields vary among different packets. OUT/IN/SETUP token packets do not have frame numbers and data. SOF toke packet does not have an address and data. Data packets do not have an address and frame number. Handshake packets only have a PID field in the packet content section.

EOP is indicated by the transmitter driving 2-bit times of SE0 and 1-bit time of J state. For more details about the bit field of every packet, refer to [245].

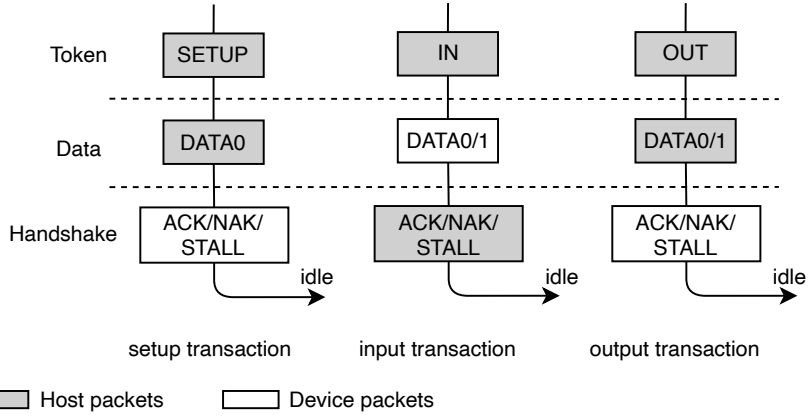


FIGURE A.10: USB transaction procedure.Texts in the blocks means the packet type. Check Table A.2 for packet detail.

Several packets make up a transaction. Packets and transactions cannot be interrupted and must be finished within a frame. All packets that belong to the same transaction should be transferred consecutively. For high-speed devices, the frame period is 125us. For full-speed and low-speed

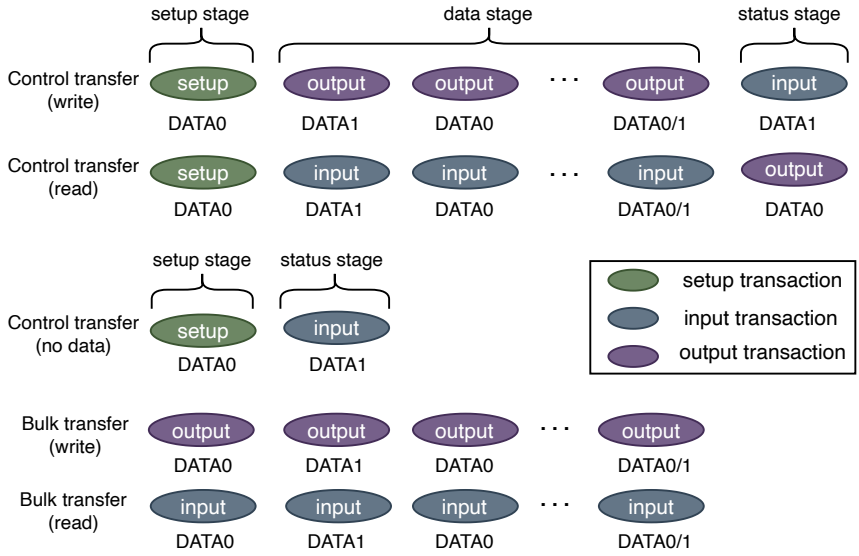


FIGURE A.11: USB setup and bulk transfer procedure.

devices, the frame period is 1ms. USB transaction has 3 types: input, output, and setup. An illustration of these three is shown in Figure A.10. All of them have three types of packets: token, data, and handshake. Two differences among them are the token packet and the data direction.

A transfer is made up of several transactions. USB has four transfer types: Bulk transfer, isochronous transfer, interrupt transfer, and control transfer. Figure A.11 illustrates two of them: control transfer and bulk transfer. Interrupt transfer is similar to bulk transfer except that it does not support PING packets. The isochronous transfer has the highest priority, so it does not require a handshake part for input and output transactions, which also means it does not have a resend mechanism. This transfer thus has the best real-time performance but does not guarantee that the data is correct. Control transfer is a little more complicated than bulk transfer. A control transfer has three stages: setup, data, and status. The setup stage has only one setup transaction. The data stage has several out/in transactions for write/read. Packet DATA0 and DATA1 are switched one by one during the data stage. No data stage for control transfer without data such as set/clear feature request [245]. The last status stage is used to mark the end of a transfer. It has the opposite direction to the data stage for write and read control transfer. For no data control transfer, the status stage is

fixed using an input transaction. No matter the transaction in the status stage is an input or output transaction, the data packets during this transaction always have zero-length data.

A.5 DAVIS346ZYNQ CONFIGURATION

```
------------------------------------------------------------------
------------------- ZYNQ DAVIS 346 CONFIG AND STATUS ------------
--------------------CASE INSENSTIVE-----------------------------
 Select one of the options below:
 ## Play mode ##
    'a' - Show play mode
    'b' - Change play mode
 ## Display mode
    'c' - Show display mode
    'd' - Change display mode
 ## event statistics
    'e' - Event rate info
    'f' - Reset max event rate
 ## ABMOF status and configuration
    'g' - Toggle ABMOF calculating mode
    'i' - Show ABMOF area event cnt threshold
    'j' - Set ABMOF area event cnt threshold
    'k' - Show ABMOF feedback mode
    'l' - Toggle ABMOF feedback mode
    'm' - Show ABMOF average target value
    'n' - Set ABMOF average target value
 ## SFAST status and configuration
    'o' - Show corner event ration
    'p' - Reset max corner ratio
    'q' - Toggle SFAST output mode
    'r' - Show SFAST threshold
    's' - Set SFAST threshold
    't' - Show SFAST area cnt threshold
 ## SD card files configuration
    'u' - Show current event number per packet
    'w' - Set current event number per packet
    'x' - Change file on the sd card to be sent
 ## Other options
     'h' - Show available keys
     'v' - Verbose Mode ON/OFF
Type your command:
```

FIGURE A.12: The configuration console of `DAVIS346Zynq`.

The interface of the IP is AXI4Lite, which is a widely used communication bus in SoC systems to interconnect modules. With this bus, the ARM on the SoC can easily access the IP at any time. The PS runs a bare-metal firmware that lets us program registers over the USB port, but it is otherwise idle. The firmware interprets the user command sent via UART and sends it to the appropriate IP using AXI4Lite. Moreover, the firmware can send DVS events to the `DAVIS346Zynq` logic from the SD card for testing. DVS events along with key points and OF are transmitted to the host computer, where we can capture them and used them to verify they are identical to the output from the software ABMOF algorithm[1].

Fig. A.12 is a screenshot of the configuration console of `DAVIS346Zynq`. It consists of seven sections. The first section is "Play mode". This camera supports two modes: live mode and playback mode. It can show the current play mode and change it at any time.

The second section is "Display mode". It supports three display modes: jAER mode, VGA mode, and mixed mode. In jAER mode, events are only sent to jAER via USB. In VGA mode, events are only displayed on a VGA monitor. For the mixed mode, events are sent to both of them simultaneously.

The third section is "Event Statistics". This mode returns the current event rate and the maximum event rate.

The fourth section is "ABMOF status and configuration". This section is used to configure ABMOF and show its running status. It can control whether to estimate the optical flow for all events or only corners. The area event number threshold k (See Table 3.1) could be adjusted by the user at any time. It also supports toggling feedback in the ABMOF.

The fifth section is "SFAST status and configuration". It can configure SFAST threshold (See Table 3.1). It supports returning the corner event statistic.

The sixth section is " SD Card configuration". It can configure how many events are used to form an event packet. The other function is to select which file for sequencing.

The last section is a miscellaneous section. It has two options: show available keys and toggle verbose mode.

1 ABMOF Java source code in jAER on GitHub (*PatchMatchFlow.java*)

A.6 HARDWARE DEBUGGING STORY

Debugging software and hardware is a totally different experience. Software debugging usually would not break hardware. Besides, it is almost possible to solve all software problems with powerful breakpoint tools. However, hardware debugging might break devices, and there are no breakpoints that could be used in hardware, especially in FPGA debugging. Most of the time, there are just some LEDs on board. When one LED is turned on, it indicates that the Verilog/VHDL code has hit some specific line. However, to use the LEDs, there is a prerequisite: the FPAG could be detected by the PC so that the PC could download the firmware to the FPGA. If we call the time that we could use a PC to communicate with the FPGA "historical time", this section tells a story of the "prehistoric time".

After receiving the assembled `DAVIS346Zynq` board, I powered on my board and connected the USB to the PC. An item was supposed to appear on the menu of the hardware target manager in Vivado. But nothing was there. The first reaction that occurred to me was to check the USB-JTAG bridge. There was an onboard USB-JTAG bridge module that could convert the Joint Test Action Group (JTAG) to USB. I had some experience with JTAG before. Nevertheless, this was the first time to use this module. For this problem, I guessed there were two possible possibilities. One was that the module was broken. The other was that the schematic designed to connect to the module might be wrong. To exclude the first possibility, I tested it on another board. The result was the same. These modules were brand-new. I then started to check the second possibility. Replacing the module required disassembling and re-soldering. There were two interfaces for JTAG wires. One was connected to the USB-JTAG bridge module, and the other was connected to a parallel interface directly. The parallel interface was a standard 14-pin interface and was widely used in many early embedded systems. But more and more recent designs adopted USB-JTAG modules because of their convenience and portability.

Fig. A.13 shows the 14-pin JTAG interface. Since the ARM and the FPGA on the Xilinx Zynq 7100 SoC share the same JTAG bus, I used an ARM JTAG simulator. There are only 5 valid pins for a 14-pin JTAG interface as shown in Fig. A.13. JTAG is usually the first interface circuit that should work after the chip is powered on. The primary purpose of JTAG is to verify the design and test the wire connections on the chip and PCB. Nowadays, it is also widely used as a debugging tool for MCUs. Most of the time, users do not need to care about the protocol behind JTAG because it

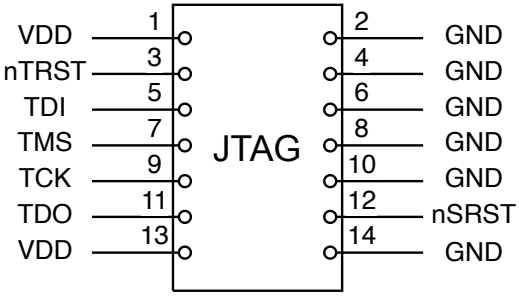


FIGURE A.13: The 14-pin interface of JTAG.

is only used as a communication tool between MCUs and PCs. The minimum requirement to enable JTAG is usually quite simple. If the PC cannot recognize the SoC, this board is a "rubbish" board. I had no choice but to try all possibilities to make it work. The first thing I did was to go through the details of the JTAG protocol and compare it with the waveforms I obtained from an oscilloscope. The waveform indeed showed something but in a short time. At last, I knew that was a waveform of the IDCODE. This IDCODE was hardcoded to the circuit register when they were fabricated. I compared the output IDCODE with the user manual of Zynq 7100. It proved that it was a valid output. The TDO pin of the JTAG could shift out the IDCODE correctly. It seemed that the output of JTAG functioned correctly. Then I did an experiment. I used a JTAG controller [2] to configure the JTAG into a bypass mode. And then, I used the controller to generate different test vectors and patterns to simulate the input pin TDI of JTAG. If everything went as I hoped, the TDO pin should shift the output as same as the input. However, there was nothing except the IDCODE. At that time, I started to doubt other circuits on the board.

According to my previous experience with the MCU, the problem should not be from the JTAG interface circuit. It is known to hardware designers that power and clock circuits are the two most essential circuits on the board. Most of the time, if the power and clock circuits work well, then it usually satisfies the minimum condition that a MCU requires. I turned my attention to these two circuits. The clock circuit was easy to check. There were two oscillators on the board. One was 24MHz used by the USB PHY circuit. We do not need to care about this. The other 50MHz clock provides

2 https://www.jtaglive.com/

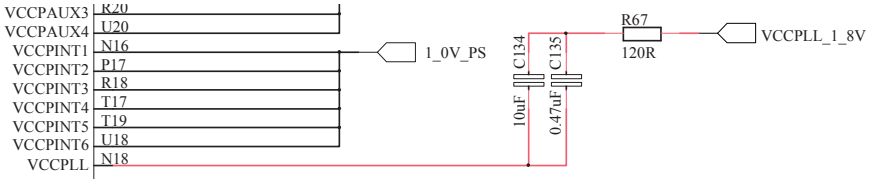


FIGURE A.14: The wrong power supply for PLL circuit of the SoC. Marked in red.

the system clock. From the oscilloscope, it can generate a very nice 50MHz square wave. The clock circuit was good.

The left possibility was to check the power circuit. The power circuit was not easy to check. As shown in Sec. 5.3.2, there were multiple power rails on the board. I decided to use an ablation study. However, before that, I need to find the minimum operation condition that an SoC could work. I had a MiniZed board on hand. Although the SoC between the MiniZed and `DAVIS346Zynq` is different, they were from the same family series. I disassembled components on the MiniZed until the PC can not recognize this board. Unexpectedly, even a clock was not required. The correct power circuit was the only condition. Then I used the power supplied by MiniZed to power on my board. My `DAVIS346Zynq` board was recognized by the PC successfully. This finding was really inspiring.

Similar to the process of finding the minimum condition, I used a combination power method to supply my board and then removed the power rail from the MiniZed one by one. Finally, I localized the error power rail. Phase-locked loops (PLL) on the chip was not powered. The error is shown in Fig. A.14. The decoupling capacitors were series-connected on the path from the power to the chip. However, they were supposed to be connected in a parallel connection. I fixed the error on the board and assembled all removed components back on the board. Actually, one SoC broke during the combination power supply test due to the wrong power-off sequence. That is why I say hardware debugging might break devices at the beginning of this section. But the good news was that the basic setup for the board was done. Both ARM and the FPGA were detected by the PC successfully. Until then, more than one month was passed.

It is a story caused by a small error. During that period, I was very desperate and did not know what I should do next. I was entirely not

sure if I can solve the problem. Sometimes, I thought I should give up. Fortunately, I did not give up and managed to solve the problem.

Follow-up stories. Of course, this problem was not the only problem. In the following time, I encountered another three significant problems: DDR3 reported many reading and writing problems, NAND flash boot problems, and the USB on the PS did not work. The DDR3 problem was caused by the super-high-speed design. DDR3 is a super-high-speed memory. To satisfy the signal's integrity, it has strict regulations on the signals' wire length. I did constrain all data signals to have an identical length. However, I ignored the control signals. This problem took me a while to fix. The straightforward solution is to redesign the board. But I am not sure if the other remaining problems on board and redesigning costs more money and more time. At last, I found a solution that did not require redesigning the board. Thanks to the on-chip DCI, it could dynamically adjust the termination resistor to compensate for the control signals' timing delay. The details of stories about the NAND flash and USB are not described here. The final solutions for them were the following: The NAND flash problem was solved by replacing it with another chip from a different company. For the USB, I did not use the USB provided by the PS and designed a customized USB controller on FPGA (See Sec. 5.3.5).

Writing the stories is fast, but the time spent behind the stories is not like writing. Luckily, I did not give up, and thus it was possible to finish the EDFLOW work. I learned three lessons from hardware debugging. One important lesson I learned from hardware debugging is that you can never check the power supply too much. The power circuit is the fundamental and core circuit for PCB design. If the power circuit dies, the whole board dies. Another lesson is to pay enough attention to all aspects of the super-high-speed signals on the board, from the schematic, the placement, and the routing. The third lesson is to extend all wires from the chip to the connector interfaces as much as possible. Like JTAG interface I used in this design, it helped me observe the waveforms from the chip directly. To sum up, hardware debugging is a painful but happy process. Through the unforgettable debugging, I also learned a lot of things about the interface circuits. I have a deeper understanding of the protocols of JTAG, DDR3, NAND flash, and USB.

A.7 DATASET AND SOURCE CODE REPOSITORY

ABMOF18 dataset data link (use Resilio Sync to download):
http://tiny.cc/htpauz

ABMOF dataset README link:
http://tiny.cc/itpauz

Petalinux project source code for npp board:
https://github.com/wzygzlm/minized_petalinux_prjs

SDx project for NPP board:
https://github.com/wzygzlm/nnp_OF_SDx

Petalinux project source code for MiniZed:
https://github.com/wzygzlm/minized_petalinux_prjs

MiniZed SDx platform files:
https://github.com/wzygzlm/mz_petalinux_SDx

jAER project:
https://github.com/SensorsINI/jaer

`DAVIS346Zynq` Vivado project, including VGA, USB, SPI:
https://github.com/wzygzlm/7z100All

EDFLOW HLS project:
https://github.com/SensorsINI/EDFLOW

BIBLIOGRAPHY

1. Mueggler, E., Rebecq, H., Gallego, G., Delbruck, T. & Scaramuzza, D. The event-camera dataset and simulator: Event-based data for pose estimation, visual odometry, and SLAM. *The International Journal of Robotics Research* **36**, 142 (2017).
2. *Phantom high speed cameras* `https://www.phantomhighspeed.com//`.
3. *Freely WAVE* `https://freeflysystems.com/wave//`.
4. *Chronos* `https://www.krontech.ca/chronos-2-1-resources//`.
5. Lichtsteiner, P., Posch, C. & Delbruck, T. A 128 x 128 120 dB 15 µs Latency Asynchronous Temporal Contrast Vision Sensor. *IEEE Journal of Solid-State Circuits* **43**, 566 (2008).
6. Liu, S.-C., Delbruck, T., Indiveri, G., Whatley, A. & Douglas, R. *Event-Based Neuromorphic Systems* 450 pp. (John Wiley and Sons Ltd., UK, 2015).
7. Delbruck, T., Linares-Barranco, B., Culurciello, E. & Posch, C. *Activity-Driven, Event-Based Vision Sensors* in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)* Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS) (Paris, 2010), 2426. doi:10.1109/ISCAS.2010.5537149.
8. Brandli, C., Berner, R., Yang, M., Liu, S.-C. & Delbruck, T. A 240x180 130 dB 3 us Latency Global Shutter Spatiotemporal Vision Sensor. *IEEE Journal of Solid-State Circuits* **49**, 2333 (2014).
9. Li, C., Brandli, C., Berner, R., Liu, H., Yang, M., Liu, S.-C. & Delbruck, T. *An RGBW Color VGA Rolling and Global Shutter Dynamic and Active-Pixel Vision Sensor* in *2015 International Image Sensor Workshop (IISW 2015)* 2015 International Image Sensor Workshop (IISW 2015) (imagesensors.org, Vaals, Netherlands, 2015), 393.
10. Posch, C., Matolin, D. & Wohlgenannt, R. *An asynchronous time-based image sensor* in *IEEE International Symposium on Circuits and Systems, 2008. ISCAS 2008* IEEE International Symposium on Circuits and Systems, 2008. ISCAS 2008 (2008), 2130. doi:10.1109/ISCAS.2008.4541871.

11. Gallego, G., Delbruck, T., Orchard, G., Bartolozzi, C., Taba, B., Censi, A., Leutenegger, S., Davison, A., Conradt, J., Daniilidis, K., *et al.* Event-based vision: A survey. *arXiv preprint arXiv:1904.08405* (2019).

12. Berner, R., Brandli, C., Yang, M., Liu, S. C. & Delbruck, T. *A 240× 180 10mW 12us latency sparse-output vision sensor for mobile applications* in *VLSI Circuits (VLSIC), 2013 Symposium on* (2013), C186.

13. Liu, S.-C., Delbruck, T., Indiveri, G., Whatley, A. & Douglas, R. *Event-Based Neuromorphic Systems* 450 pp. (John Wiley and Sons Ltd., UK, 2015).

14. Gallego, G., Delbruck, T., Orchard, G. M., Bartolozzi, C., Taba, B., Censi, A., Leutenegger, S., Davison, A., Conradt, J., Daniilidis, K. & Scaramuzza, D. Event-based Vision: A Survey. *IEEE Trans. Pattern Anal. Mach. Intell.* **PP**, 1 (2020).

15. Chin, T.-J., Bagchi, S., Eriksson, A. & van Schaik, A. *Star Tracking Using an Event Camera* in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops* (2019).

16. Bagchi, S. & Chin, T.-J. *Event-based star tracking via multiresolution progressive Hough transforms* in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision* (2020), 2143. doi:10.1109/WACV45572.2020.9093309.

17. Vasco, V., Glover, A. & Bartolozzi, C. *Fast event-based Harris corner detection exploiting the advantages of event-driven cameras* in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2016), 4144.

18. Mueggler, E., Bartolozzi, C. & Scaramuzza, D. *Fast event-based corner detection* in *British Machine Vis. Conf.(BMVC)* **1** (2017).

19. Alzugaray, I. & Chli, M. Asynchronous corner detection and tracking for event cameras in real time. *IEEE Robotics and Automation Letters* **3**, 3177 (2018).

20. Manderscheid, J., Sironi, A., Bourdis, N., Migliore, D. & Lepetit, V. Speed Invariant Time Surface for Learning to Detect Corner Points with Event-Based Cameras. *arXiv preprint arXiv:1903.11332* (2019).

21. Stoffregen, T. & Kleeman, L. Simultaneous Optical Flow and Segmentation (SOFAS) using Dynamic Vision Sensor (2018).

22. Rebecq, H., Gehrig, D. & Scaramuzza, D. *Esim: an open event camera simulator* in *Conference on Robot Learning* (2018), 969.

23. Hu, Y., Liu, S.-C. & Delbruck, T. *v2e: From Video Frames to Realistic DVS Events* in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops* (2021), 1312.

24. Zhu, A. Z., Thakur, D., Özaslan, T., Pfrommer, B., Kumar, V. & Daniilidis, K. The multivehicle stereo event camera dataset: An event camera dataset for 3D perception. *IEEE Robotics and Automation Letters* **3**, 2032 (2018).

25. Binas, J., Neil, D., Liu, S.-C. & Delbruck, T. DDD17: End-to-end DAVIS driving dataset. *arXiv preprint arXiv:1711.01458* (2017).

26. Scheerlinck, C., Rebecq, H., Stoffregen, T., Barnes, N., Mahony, R. & Scaramuzza, D. *CED: Color event camera dataset* in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops* (2019), 0.

27. Calabrese, E., Taverni, G., Awai Easthope, C., Skriabine, S., Corradi, F., Longinotti, L., Eng, K. & Delbruck, T. *DHP19: Dynamic Vision Sensor 3D Human Pose Dataset* in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops* (2019), 0.

28. Gallego, G. & Scaramuzza, D. Accurate angular velocity estimation with an event camera. *IEEE Robotics and Automation Letters* **2**, 632 (2017).

29. Zhu, A. Z., Atanasov, N. & Daniilidis, K. *Event-based feature tracking with probabilistic data association* in *Robotics and Automation (ICRA), 2017 IEEE International Conference on* (2017), 4465.

30. Hadviger, A., Marković, I. & Petrović, I. Stereo dense depth tracking based on optical flow using frames and events. *Advanced Robotics*, 1 (2020).

31. McGuire, K., De Croon, G., De Wagter, C., Tuyls, K. & Kappen, H. Efficient optical flow and stereo vision for velocity estimation and obstacle avoidance on an autonomous pocket drone. *IEEE Robotics and Automation Letters* **2**, 1070 (2017).

32. De Croon, G., Ho, H., De Wagter, C., Van Kampen, E., Remes, B. & Chu, Q. Optic-flow based slope estimation for autonomous landing. *International Journal of Micro Air Vehicles* **5**, 287 (2013).

33. Hordijk, B. J. P., Scheper, K. Y. & de Croon, G. G. Vertical Landing for Micro Air Vehicles using Event-Based Optical Flow. *arXiv preprint arXiv:1702.00061* (2017).

34. Pijnacker Hordijk, B. J., Scheper, K. Y. & De Croon, G. C. Vertical landing for micro air vehicles using event-based optical flow. *Journal of Field Robotics* **35**, 69 (2018).

35. Weikersdorfer, D. & Conradt, J. *Event-based particle filtering for robot self-localization* in *Robotics and Biomimetics (ROBIO), 2012 IEEE International Conference on* (2012), 866.

36. Zingg, S., Scaramuzza, D., Weiss, S. & Siegwart, R. *MAV navigation through indoor corridors using optical flow* in *2010 IEEE International Conference on Robotics and Automation* (2010), 3361. doi:10.1109/robot.2010.5509777.

37. Mueggler, E., Gallego, G. & Scaramuzza, D. *Continuous-time trajectory estimation for event-based vision sensors* in *Robotics: Science and Systems XI* (2015).

38. Gallego, G., Lund, J. E., Mueggler, E., Rebecq, H., Delbruck, T. & Scaramuzza, D. Event-based, 6-DOF camera tracking for high-speed applications. *arXiv preprint arXiv:1607.03468* (2016).

39. Zhu, A. Z., Atanasov, N. & Daniilidis, K. Event-based Visual Inertial Odometry (2017).

40. Kueng, B., Mueggler, E., Gallego, G. & Scaramuzza, D. *Low-latency visual odometry using event-based feature tracks* in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (2016), 16. doi:10.1109/IROS.2016.7758089.

41. Censi, A. & Scaramuzza, D. *Low-latency event-based visual odometry* in *2014 IEEE International Conference on Robotics and Automation (ICRA)* 2014 IEEE International Conference on Robotics and Automation (ICRA) (2014), 703. doi:10.1109/ICRA.2014.6906931.

42. Rebecq, H., Horstschaefer, T. & Scaramuzza, D. *Real-time visual-inertial odometry for event cameras using keyframe-based nonlinear optimization* in *British Machine Vis. Conf.(BMVC)* **3** (2017).

43. Weikersdorfer, D., Hoffmann, R. & Conradt, J. *Simultaneous localization and mapping for event-based vision systems* in *International Conference on Computer Vision Systems* (2013), 133.

44. Vidal, A. R., Rebecq, H., Horstschaefer, T. & Scaramuzza, D. Ultimate SLAM? Combining events, images, and IMU for robust visual SLAM in HDR and high-speed scenarios. *IEEE Robotics and Automation Letters* **3**, 994 (2018).

45. Gibson, J. J. *The ecological approach to visual perception: classic edition* (Psychology Press, 2014).

46. Torii, A., Imiya, A., Sugaya, H. & Mochizuki, Y. *Optical flow computation for compound eyes: Variational analysis of omni-directional views* in *International Symposium on Brain, Vision, and Artificial Intelligence* (2005), 527.

47. Chahl, J. S., Srinivasan, M. V. & Zhang, S.-W. Landing strategies in honeybees and applications to uninhabited airborne vehicles. *The International Journal of Robotics Research* **23**, 101 (2004).

48. Baird, E., Boeddeker, N., Ibbotson, M. R. & Srinivasan, M. V. A universal strategy for visually guided landing. *Proceedings of the National Academy of Sciences* **110**, 18686 (2013).

49. Esch, H. & Burns, J. Distance estimation by foraging honeybees. *Journal of Experimental Biology* **199**, 155 (1996).

50. Srinivasan, M. V. Honeybees as a model for the study of visually guided flight, navigation, and biologically inspired robotics. *Physiological reviews* **91**, 413 (2011).

51. Schuster, R., Bailer, C., Wasenmüller, O. & Stricker, D. in *Commercial Vehicle Technology 2018* 90 (Springer, 2018).

52. Yin, Z. & Shi, J. *Geonet: Unsupervised learning of dense depth, optical flow and camera pose* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2018), 1983.

53. Jain, M., Jégou, H. & Bouthemy, P. *Better exploiting motion for better action recognition* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2013), 2555.

54. Wang, H., Kläser, A., Schmid, C. & Liu, C.-L. Dense trajectories and motion boundary descriptors for action recognition. *International journal of computer vision* **103**, 60 (2013).

55. Jakubowski, M. & Pastuszak, G. Block-based motion estimation algorithms — a survey. *Opto-Electronics Review* **21**, 86 (2013).

56. Cheng, J., Tsai, Y.-H., Wang, S. & Yang, M.-H. *Segflow: Joint learning for video object segmentation and optical flow* in *Proceedings of the IEEE international conference on computer vision* (2017), 686.

57. Ding, M., Wang, Z., Zhou, B., Shi, J., Lu, Z. & Luo, P. *Every frame counts: joint learning of video segmentation and optical flow* in *Proceedings of the AAAI Conference on Artificial Intelligence* **34** (2020), 10713.

58. Geiger, A., Lenz, P. & Urtasun, R. *Are we ready for autonomous driving? the kitti vision benchmark suite* in *2012 IEEE Conference on Computer Vision and Pattern Recognition* (2012), 3354.

59. Chao, H., Gu, Y. & Napolitano, M. A survey of optical flow techniques for robotics navigation applications. *Journal of Intelligent & Robotic Systems* **73**, 361 (2014).

60. Crétual, A. & Chaumette, F. Visual servoing based on image motion. *The International Journal of Robotics Research* **20**, 857 (2001).

61. Giachetti, A., Campani, M. & Torre, V. The use of optical flow for road navigation. *IEEE transactions on robotics and automation* **14**, 34 (1998).

62. Sun, Z., Bebis, G. & Miller, R. On-road vehicle detection: A review. *IEEE transactions on pattern analysis and machine intelligence* **28**, 694 (2006).

63. Enkelmann, W. Obstacle detection by evaluation of optical flow fields from image sequences. *Image and Vision Computing* **9**, 160 (1991).

64. Miller, A., Miller, B., Popov, A. & Stepanyan, K. UAV landing based on the optical flow videonavigation. *Sensors* **19**, 1351 (2019).

65. Cheng, H.-W., Chen, T.-L. & Tien, C.-H. Motion estimation by hybrid optical flow technology for UAV landing in an unvisited area. *Sensors* **19**, 1380 (2019).

66. Wang, Z., Wang, B., Tang, C. & Xu, G. *Pose and Velocity Estimation Algorithm for UAV in Visual Landing* in *2020 39th Chinese Control Conference (CCC)* (2020), 3713.

67. Horn, B. K. & Schunck, B. G. Determining optical flow. *Artificial intelligence* **17**, 185 (1981).

68. Lucas, B. D., Kanade, T., *et al.* An iterative image registration technique with an application to stereo vision (1981).

69. Black, M. J. & Anandan, P. The robust estimation of multiple motions: Parametric and piecewise-smooth flow fields. *Computer vision and image understanding* **63**, 75 (1996).

70. Brox, T., Bregler, C. & Malik, J. *Large displacement optical flow* in *2009 IEEE Conference on Computer Vision and Pattern Recognition* (2009), 41.

71. Mileva, Y., Bruhn, A. & Weickert, J. *Illumination-robust variational optical flow with photometric invariants* in *Joint Pattern Recognition Symposium* (2007), 152.

72. Zimmer, H., Bruhn, A., Weickert, J., Valgaerts, L., Salgado, A., Rosenhahn, B. & Seidel, H.-P. *Complementary optic flow* in *International Workshop on Energy Minimization Methods in Computer Vision and Pattern Recognition* (2009), 207.

73. Kerfa, D. & Belbachir, M. F. Star diamond: an efficient algorithm for fast block matching motion estimation in H264/AVC video codec. *Multimedia tools and applications* **75**, 3161 (2016).

74. Lu, J. & Liou, M. L. A simple and efficient search algorithm for block-matching motion estimation. *IEEE Transactions on Circuits and Systems for Video Technology* **7**, 429 (1997).

75. Li, R., Zeng, B. & Liou, M. L. A new three-step search algorithm for block motion estimation. *IEEE transactions on circuits and systems for video technology* **4**, 438 (1994).

76. Zhu, S. & Ma, K.-K. A new diamond search algorithm for fast block-matching motion estimation. *IEEE transactions on Image Processing* **9**, 287 (2000).

77. Nie, Y. & Ma, K.-K. Adaptive rood pattern search for fast block-matching motion estimation. *IEEE Trans. Image Process.* **11**, 1442 (2002).

78. Black, M. J. *Robust incremental optical flow* PhD thesis (Verlag nicht ermittelbar, 1992).

79. Brox, T., Bruhn, A., Papenberg, N. & Weickert, J. *High accuracy optical flow estimation based on a theory for warping* in *European conference on computer vision* (2004), 25.

80. Weinzaepfel, P., Revaud, J., Harchaoui, Z. & Schmid, C. *DeepFlow: Large displacement optical flow with deep matching* in *Proceedings of the IEEE international conference on computer vision* (2013), 1385.

81. Revaud, J., Weinzaepfel, P., Harchaoui, Z. & Schmid, C. *Epicflow: Edge-preserving interpolation of correspondences for optical flow* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), 1164.

82. Dosovitskiy, A., Fischer, P., Ilg, E., Hausser, P., Hazirbas, C., Golkov, V., Van Der Smagt, P., Cremers, D. & Brox, T. *Flownet: Learning optical flow with convolutional networks* in *Proceedings of the IEEE international conference on computer vision* (2015), 2758.

83. Ilg, E., Mayer, N., Saikia, T., Keuper, M., Dosovitskiy, A. & Brox, T. *Flownet 2.0: Evolution of optical flow estimation with deep networks* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017), 2462.

84. Hui, T.-W., Tang, X. & Loy, C. C. *Liteflownet: A lightweight convolutional neural network for optical flow estimation* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2018), 8981.

85. Hui, T.-W., Tang, X. & Loy, C. C. A lightweight optical flow CNN Revisiting data fidelity and regularization. *IEEE transactions on pattern analysis and machine intelligence* **43**, 2555 (2020).

86. Ranjan, A. & Black, M. J. *Optical flow estimation using a spatial pyramid network* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017), 4161.

87. Sun, D., Yang, X., Liu, M.-Y. & Kautz, J. *Pwc-net: Cnns for optical flow using pyramid, warping, and cost volume* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2018), 8934.

88. Savian, S., Elahi, M. & Tillo, T. in *Deep biometrics* 257 (Springer, 2020).

89. Fang, M., Li, Y., Han, Y. & Wen, J. *A deep convolutional network based supervised coarse-to-fine algorithm for optical flow measurement* in *2018 IEEE 20th International Workshop on Multimedia Signal Processing (MMSP)* (2018), 1.

90. Baker, S., Scharstein, D., Lewis, J., Roth, S., Black, M. J. & Szeliski, R. A database and evaluation methodology for optical flow. *International Journal of Computer Vision* **92**, 1 (2011).

91. Butler, D. J., Wulff, J., Stanley, G. B. & Black, M. J. *A naturalistic open source movie for optical flow evaluation* in *European conference on computer vision* (2012), 611.

92. Geiger, A., Lenz, P., Stiller, C. & Urtasun, R. Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research* **32**, 1231 (2013).

93. Menze, M. & Geiger, A. *Object scene flow for autonomous vehicles* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), 3061.

94. Mathur, R. Evaluation Datasets and Benchmarks for Optical Flow Algorithms: A Review (2020).

95. Schröder, G., Senst, T., Bochinski, E. & Sikora, T. *Optical flow dataset and benchmark for visual crowd analysis* in *2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)* (2018), 1.

96. Mayer, N., Ilg, E., Hausser, P., Fischer, P., Cremers, D., Dosovitskiy, A. & Brox, T. *A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), 4040.

97. Shugrina, M., Liang, Z., Kar, A., Li, J., Singh, A., Singh, K. & Fidler, S. *Creative flow+ dataset* in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2019), 5384.

98. Fortun, D., Bouthemy, P. & Kervrann, C. Optical flow modeling and computation: A survey. *Computer Vision and Image Understanding* **134**, 1 (2015).

99. Hyneman, W., Itokazu, H., Williams, L. & Zhao, X. *Human Face Project* in *ACM SIGGRAPH 2005 Courses* (Association for Computing Machinery, Los Angeles, California, 2005), 5. doi:10.1145/1198555.1198585.

100. Floreano, D. & Wood, R. J. Science, technology and the future of small autonomous drones. *Nature* **521**, 460 (2015).

101. Zufferey, J.-C., Beyeler, A. & Floreano, D. Autonomous flight at low altitude using light sensors and little computational power. *International Journal of Micro Air Vehicles* **2**, 107 (2010).

102. Fraundorfer, F., Heng, L., Honegger, D., Lee, G. H., Meier, L., Tanskanen, P. & Pollefeys, M. *Vision-based autonomous mapping and exploration using a quadrotor MAV* in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2012), 4557.

103. Ross, R., Devlin, J. & Wang, S. Toward refocused optical mouse sensors for outdoor optical flow odometry. *IEEE Sensors Journal* **12**, 1925 (2011).

104. Pallejà, T., Soler, E. R., Teixidó, M., Tresanchez, M., Del Viso, A. F., Sánchez, C. R. & Palacin, J. Using the optical flow to implement a relative virtual mouse controlled by head movements. *J. UCS* **14**, 3127 (2008).

105. Honegger, D., Meier, L., Tanskanen, P. & Pollefeys, M. *An open source and open hardware embedded metric optical flow cmos camera for indoor and outdoor applications* in *2013 IEEE International Conference on Robotics and Automation* (2013), 1736.

106. Kendoul, F., Fantoni, I. & Nonami, K. Optic flow-based vision system for autonomous 3D localization and control of small aerial vehicles. *Robotics and autonomous systems* **57**, 591 (2009).

107. Griffith, S., Saunders, J., Curtis, A., Barber, B., McLain, T. & Beard, R. Maximizing miniature aerial vehicles—Obstacle and terrain avoidance for MAVs. *IEEE Robotics & Automation Magazine* **13**, 34 (2006).

108. Watman, D. & Murayama, H. *Design of a miniature, multi-directional optical flow sensor for micro aerial vehicles* in *2011 IEEE International Conference on Robotics and Automation* (2011), 2986.

109. Chao, H., Gu, Y., Gross, J., Guo, G., Fravolini, M. L. & Napolitano, M. R. *A comparative study of optical flow and traditional sensors in uav navigation* in *2013 American Control Conference* (2013), 3858.

110. Floreano, D., Zufferey, J.-C., Srinivasan, M. V. & Ellington, C. *Flying insects and robots* (Springer, 2009).

111. Moini, A. *Vision chips* (Springer Science & Business Media, 1999).

112. Mead, C. *Analog VLSI and neural systems* (Addison-Wesley Longman Publishing Co., Inc., 1989).

113. Stocker, A. A. *Analog VLSI circuits for the perception of visual motion* (Wiley Online Library, 2006).

114. Liu, M. & Delbruck, T. *Block-matching optical flow for dynamic vision sensors: Algorithm and FPGA implementation* in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)* (2017), 1. doi:10.1109/ISCAS.2017.8050295.

115. Rueckauer, B. & Delbruck, T. Evaluation of event-based algorithms for optical flow with ground-truth from inertial measurement sensor. *Frontiers in neuroscience* **10**, 176 (2016).

116. Gallego, G., Rebecq, H. & Scaramuzza, D. *A unifying contrast maximization framework for event cameras, with applications to motion, depth, and optical flow estimation* in (Salt City, 2018).

117. Delbruck, T. *Frame-free dynamic digital vision* in *Proceedings of Intl. Symp. on Secure-Life Electronics, Advanced Electronics for Quality Life and Society* (2008), 21.

118. Benosman, R., Ieng, S.-H., Clercq, C., Bartolozzi, C. & Srinivasan, M. Asynchronous frameless event-based optical flow. *Neural Networks* **27**, 32 (2012).

119. Barranco, F., Fermüller, C. & Aloimonos, Y. Contour motion estimation for asynchronous event-driven cameras. *Proceedings of the IEEE* **102**, 1537 (2014).

120. Benosman, R., Clercq, C., Lagorce, X., Ieng, S.-H. & Bartolozzi, C. Event-based visual flow. *IEEE Trans Neural Netw Learn Syst* **25**, 407 (2014).

121. Zhu, A. Z., Yuan, L., Chaney, K. & Daniilidis, K. EV-FlowNet: Self-Supervised Optical Flow Estimation for Event-based Cameras. *arXiv:1802.06898 [cs]* (2018).

122. Barranco, F., Fermuller, C. & Aloimonos, Y. *Bio-inspired motion estimation with event-driven sensors* in *International Work-Conference on Artificial Neural Networks* (2015), 309.

123. Bardow, P., Davison, A. J. & Leutenegger, S. *Simultaneous optical flow and intensity estimation from an event camera* in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), 884.

124. Gallego, G., Gehrig, M. & Scaramuzza, D. *Focus is all you need: Loss functions for event-based vision* in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (openaccess.thecvf.com, 2019), 12280.

125. Stoffregen, T. & Kleeman, L. *Event cameras, contrast maximization and reward functions: an analysis* in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2019), 12300. doi:10.1109/cvpr.2019.01258.

126. Pan, L., Liu, M. & Hartley, R. *Single Image Optical Flow Estimation With an Event Camera* in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2020), 1672.

127. Akolkar, H., Ieng, S. H. & Benosman, R. Real-time high speed motion prediction using fast aperture-robust event-driven visual flow. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. doi:10.1109/tpami.2020.3010468 (2020).

128. Fei Low, W., Gao, Z., Xiang, C. & Ramesh, B. *SOFEA: A Non-Iterative and Robust Optical Flow Estimation Algorithm for Dynamic Vision Sensors* in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops* (2020), 82. doi:10.1109/cvpr42600.2020.00174.

129. Zhu, A., Yuan, L., Chaney, K. & Daniilidis, K. *EV-FlowNet: Self-Supervised Optical Flow Estimation for Event-based Cameras* in *Proceedings of Robotics: Science and Systems* (Pittsburgh, Pennsylvania, 2018). doi:10.15607/RSS.2018.XIV.062.

130. Ye, C., Mitrokhin, A., Fermüller, C., Yorke, J. A. & Aloimonos, Y. Unsupervised Learning of Dense Optical Flow, Depth and Egomotion from Sparse Event Data (2018).

131. Zhu, A. Z., Yuan, L., Chaney, K. & Daniilidis, K. *Unsupervised event-based learning of optical flow, depth, and egomotion* in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (openaccess.thecvf.com, 2019), 989. doi:10.1109/cvpr.2019.00108.

132. Lee, C., Kosta, A. K., Zhu, A. Z., Chaney, K., Daniilidis, K. & Roy, K. *Spike-FlowNet: Event-based optical flow estimation with energy-efficient hybrid neural networks* in *European Conference on Computer Vision* (2020), 366. doi:10.1007/978-3-030-58526-6_22.

133. Zihao Zhu, A., Yuan, L., Chaney, K. & Daniilidis, K. *Unsupervised Event-Based Learning of Optical Flow, Depth, and Egomotion* in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2019), 989.

134. Ye, C., Mitrokhin, A., Fermüller, C., Yorke, J. A. & Aloimonos, Y. *Unsupervised Learning of Dense Optical Flow, Depth and Egomotion from Sparse Event Data* in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2020), 5831. doi:10.1109/IROS45743.2020.9341224.

135. Paredes-Vallés, F., Scheper, K. Y. & de Croon, G. C. Unsupervised learning of a hierarchical spiking neural network for optical flow estimation: From events to global motion perception. *IEEE transactions on pattern analysis and machine intelligence* **42**, 2051 (2019).

136. Conradt, J. *On-board real-time optic-flow for miniature event-based vision sensors* in *2015 IEEE International Conference on Robotics and Biomimetics (ROBIO)* (2015), 1858.

137. Agha, S. & Dwyer, V. M. Algorithms and VLSI architectures for MPEG-4 motion estimation. *Electronic systems and control Division Research*, 24 (2003).

138. Berner, R., Delbruck, T., Civit-Balcells, A. & Linares-Barranco, A. *A 5 Meps $100 USB2.0 address-event monitor-sequencer interface* in *2007 IEEE International Symposium on Circuits and Systems* (2007), 2451.

139. *PatchMatchFlow java source code* `https:jaerproject.net`https:jaerproject.net.

140. Barjatya, A. Block matching algorithms for motion estimation. *IEEE Transactions Evolution Computation* **8**, 225 (2004).

141. Zhang, L., Zhang, Y., Tang, J., Lu, K. & Tian, Q. *Binary code ranking with weighted hamming distance* in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2013), 1586.

142. Liu, M. & Delbruck, T. *Adaptive Time-Slice Block-Matching Optical Flow Algorithm for Dynamic Vision Sensors* in *British Machine Vision Conference (BMVC) 2018* BMVC 2018 (Nescatle upon Tyne, 2018), 12.

143. Zhu, S. & Ma, K.-K. *A new diamond search algorithm for fast block matching motion estimation* in *Information, Communications and Signal Processing, 1997. ICICS., Proceedings of 1997 International Conference on* **1** (1997), 292.

144. Lindeberg, T. Scale-space theory: A basic tool for analyzing structures at different scales. *Journal of applied statistics* **21**, 225 (1994).

145. *PatchMatchFlow java source code* `https://sourceforge.net/p/jaer/code/HEAD/tree/jAER/trunk/src/ch/unizh/ini/jaer/projects/minliu/PatchMatchFlow.java//`.

146. Delbruck, T. *Java AER Framework* `https://jaerproject.org`.

147. Shi, J. *et al. Good features to track* in *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on* (1994), 593.

148. *OpenCV: Optical Flow*

149. PixArt Imaging Inc. *PixArt PMW3360 Optical Gaming Navigation Sensor* 2014.

150. Vidal, A., Rebecq, H., Horstschaefer, T. & Scaramuzza, D. Ultimate SLAM? Combining Events, Images, and IMU for Robust Visual SLAM in HDR and High Speed Scenarios. *Robotics and Autonomous Letters* (2018).

151. Moeys, D. P. *et al. Steering a predator robot using a mixed frame/event-driven convolutional neural network* in *2016 Second International Conference on Event-based Control, Communication, and Signal Processing (EBCCSP)* (2016), 1. doi:10.1109/EBCCSP.2016.7605233.

152. Lungu, I.-A., Corradi, F. & Delbruck, T. *Live Demonstration: Convolutional Neural Network Driven by Dynamic Vision Sensor Playing RoShamBo* in *2017 IEEE Symposium on Circuits and Systems (ISCAS 2017)* (Baltimore, MD, USA, 2017).

153. Amir, A. *et al. A Low Power, Fully Event-Based Gesture Recognition System* in (2017), 7243.

154. Fischl, K. D. *et al. Neuromorphic self-driving robot with retinomorphic vision and spike-based processing/closed-loop control* in *2017 51st Annual Conference on Information Sciences and Systems (CISS)* (2017), 1. doi:10.1109/CISS.2017.7926179.

155. Liu, M., Kao, W.-T. & Delbruck, T. *Live Demonstration: A Real-Time Event-Based Fast Corner Detection Demo Based on FPGA* in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)* (2019), 1678. doi:10.1109/CVPRW.2019.00212.

156. Clady, X., Ieng, S.-H. & Benosman, R. Asynchronous event-based corner detection and matching. *Neural Networks* **66**, 91 (2015).

157. Li, J., Guo, C., Su, L., Wang, X. & Hu, Q. SE-Harris and eSUSAN: Asynchronous Event-Based Corner Detection Using Megapixel Resolution CeleX-V Camera (2021).

158. Xilinx, S. *SDSoC environment Profiling and Optimization Guide* 2018.

159. Xilinx, S. *SDSoC environment User Guide* 2018.

160. Xilinx, V.-H. *Vivado Design Suite User Guide-High-Level Synthesis* 2018.

161. Kastner, R., Matai, J. & Neuendorffer, S. Parallel programming for fpgas. *arXiv preprint arXiv:1805.03648* (2018).

162. Licht, J. d. F., Meierhans, S. & Hoefler, T. Transformations of High-Level Synthesis Codes for High-Performance Computing. *arXiv preprint arXiv:1805.08288* (2018).

163. George, N., Lee, H., Novo, D., Rompf, T., Brown, K. J., Sujeeth, A. K., Odersky, M., Olukotun, K. & Ienne, P. *Hardware system synthesis from domain-specific languages* in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)* (2014), 1.

164. Matai, J., Richmond, D., Lee, D. & Kastner, R. Enabling FPGAs for the masses. *arXiv preprint arXiv:1408.5870* (2014).

165. Xilinx, S. *7 Series FPGA Memory Resources, v1.13* 2019.

166. Abdelhadi, A. & Lemieux, G. G. *Modular multi-ported SRAM-based memories* in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays* (2014), 35.

167. Laforest, C. E., Li, Z., O'rourke, T., Liu, M. G. & Steffan, J. G. Composing multi-ported memories on FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* **7**, 16 (2014).

168. Boahen, K. A. Point-to-point connectivity between neuromorphic chips using address events. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* **47**, 416 (2000).

169. Berner, R. *Building-blocks for event-based vision sensors* PhD thesis (ETH Zurich, 2011).

170. Suh, Y., Choi, S., Ito, M., Kim, J., Lee, Y., Seo, J., Jung, H., Yeo, D.-H., Namgung, S., Bong, J., *et al. A 1280× 960 Dynamic Vision Sensor with a 4.95-µm Pixel Pitch and Motion Artifact Minimization* in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)* (2020), 1. doi:10.1109/ISCAS45731.2020.9180436.

171. Rivas, M., Gomez-Rodriguez, F., Paz, R., Linares-Barranco, A., Vicente, S. & Cascado, D. *Tools for address-event-representation communication systems and debugging* in *International Conference on Artificial Neural Networks* (2005), 289.

172. Rios-Navarro, A., Dominguez-Morales, J., Tapiador-Morales, R., Gutierrez-Galan, D., Jimenez-Fernandez, A. & Linares-Barranco, A. *A 20Mevps/32Mev event-based USB framework for neuromorphic systems debugging* in *2016 Second International Conference on Event-based Control, Communication, and Signal Processing (EBCCSP)* (2016), 1.

173. Linares-Barranco, A., Gomez-Rodriguez, F., Villanueva, V., Longinotti, L. & Delbruck, T. *A USB3. 0 FPGA event-based filtering and tracking framework for dynamic vision sensors* in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)* (2015), 2417.

174. Schuman, C. D., Potok, T. E., Patton, R. M., Birdwell, J. D., Dean, M. E., Rose, G. S. & Plank, J. S. A survey of neuromorphic computing and neural networks in hardware. *arXiv preprint arXiv:1705.06963* (2017).

175. Linares-Barranco, A., Perez-Peña, F., Moeys, D. P., Gomez-Rodriguez, F., Jimenez-Moreno, G., Liu, S.-C. & Delbruck, T. Low Latency Event-Based Filtering and Feature Extraction for Dynamic Vision Sensors in Real-Time FPGA Applications. *IEEE Access* **7**, 134926 (2019).

176. Iakymchuk, T., Rosado, A., Serrano-Gotarredona, T., Linares-Barranco, B., Jiménez-Fernandez, A., Linares-Barranco, A. & Jiménez-Moreno, G. *An AER handshake-less modular infrastructure PCB with x8 2.5 Gbps LVDS serial links* in *2014 IEEE International Symposium on Circuits and Systems (ISCAS)* (2014), 1556. doi:10.1109/iscas.2014.6865445.

177. Taverni, G., Moeys, D. P., Li, C., Cavaco, C., Motsnyi, V., Bello, D. S. S. & Delbruck, T. *Front and Back Illuminated Dynamic and Active Pixel Vision Sensors Comparison* in *2018 IEEE International Symposium on Circuits and Systems (ISCAS) (accepted)* (Florence, Italy, 2018).

178. Xilinx, S. *Zynq-7000 SoC DC and AC Switching Characteristics* 2018.

179. Xilinx, S. *Zynq-7000 SoC Technology Reference Manual* 2018.

180. Sivilotti, M. A. *Wiring considerations in analog VLSI systems, with application to field-programmable networks.*

181. Liu, S.-C., Van Schaik, A., Minch, B. A. & Delbruck, T. *Event-based 64-channel binaural silicon cochlea with Q enhancement mechanisms* in *2010 IEEE International Symposium on Circuits and Systems (ISCAS)* (2010), 2027.

182. Moradi, S., Qiao, N., Stefanini, F. & Indiveri, G. A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (DYNAPs). *IEEE transactions on biomedical circuits and systems* **12**, 106 (2017).

183. Qiao, N., Mostafa, H., Corradi, F., Osswald, M., Stefanini, F., Sumislawska, D. & Indiveri, G. A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128K synapses. *Frontiers in neuroscience* **9**, 141 (2015).

184. Berner, R., Delbruck, T., Civit-Balcells, A. & Linares-Barranco, A. *A 5 MEPs $100 USB2.0 address-event monitor-sequencer interface* in *2007 IEEE International Symposium on Circuits and Systems* (IEEE, New Orleans, LA, USA, 2007), 2451. doi:10.1109/iscas.2007.378616.

185. Mead, C. A. & Delbrück, T. Scanners for visualizing activity of analog VLSI circuitry. *Analog Integrated Circuits and Signal Processing* **1**, 93 (1991).

186. Specification, U. S. B. *USB 2.0 Transceiver Macrocell Interface (UTMI) Specification* 2001.

187. Specification, U. S. B. *UTMI+ Low Pin Interface (ULPI) Specification* 2004.

188. Hu, Y., Binas, J., Neil, D., Liu, S.-C. & Delbruck, T. *DDD20 End-to-End Event Camera Driving Dataset: Fusing Frames and Events with Deep Learning for Improved Steering Prediction* in *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)* (ieeexplore.ieee.org, 2020), 1. doi:10.1109/ITSC45102.2020.9294515.

189. Tun Aung, M., Teo, R. & Orchard, G. *Event-based Plane-fitting Optical Flow for Dynamic Vision Sensors in FPGA* in *IEEE Int. Symp. Circuits Syst. (ISCAS)* (Florence, Italy, 2018).

190. Huang, J., Guo, M., Wang, S. & Chen, S. *A motion sensor with on-chip pixel rendering module for optical flow gradient extraction* in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)* (2018), 1. doi:10.1109/iscas.2018.8351312.

191. Haessig, G., Cassidy, A., Alvarez, R., Benosman, R. & Orchard, G. Spiking Optical Flow for Event-Based Sensors Using IBM's TrueNorth Neurosynaptic System. *IEEE Trans. Biomed. Circuits Syst.* **12**, 860 (2018).

192. Barlow, H. B. & Levick, W. R. The mechanism of directionally selective units in rabbit's retina. *J. Physiol.* **178**, 477 (1965).

193. Seyid, K., Richaud, A., Capoccia, R. & Leblebici, Y. FPGA-Based Hardware Implementation of Real-Time Optical Flow Calculation. *IEEE Trans. Circuits Syst. Video Technol.* **28**, 206 (2018).

194. Correia, M. V. & Campilho, A. C. *Real-time implementation of an optical flow algorithm* in *Proc. 16th Int. Conf. Pattern Recognition* **4** (ieeexplore.ieee.org, 2002), 247. doi:10.1109/ICPR.2002.1047443.

195. Campilho, A. C. *Real-Time Implementation of an Optical Flow Algorithm* in *Proceedings of the 16th International Conference on Pattern Recognition (ICPR'02) Volume 4 - Volume 4* (IEEE Computer Society, USA, 2002), 40247. doi:10.1109/icpr.2002.1047443.

196. Diaz, J., Ros, E., Pelayo, F., Ortigosa, E. M. & Mota, S. FPGA-based real-time optical-flow system. *IEEE Trans. Circuits Syst. Video Technol.* **16**, 274 (2006).

197. Mahalingam, V., Bhattacharya, K., Ranganathan, N., Chakravarthula, H., Murphy, R. R. & Pratt, K. S. A VLSI Architecture and Algorithm for Lucas–Kanade-Based Optical Flow Computation. *IEEE Trans. Very Large Scale Integr. VLSI Syst.* **18**, 29 (2010).

198. Wei, Z., Lee, D.-J., Nelson, B. E. & Archibald, J. K. Hardware-Friendly Vision Algorithms for Embedded Obstacle Detection Applications. *IEEE Trans. Circuits Syst. Video Technol.* **20**, 1577 (2010).

199. Diaz, J., Ros, E., Agis, R. & Bernier, J. L. Superpipelined high-performance optical-flow computation architecture. *Comput. Vis. Image Underst.* **112**, 262 (2008).

200. Seong, H.-S., Rhee, C. E. & Lee, H.-J. A Novel Hardware Architecture of the Lucas–Kanade Optical Flow for Reduced Frame Memory Access. *IEEE Trans. Circuits Syst. Video Technol.* **26**, 1187 (2016).

201. Kunz, M., Ostrowski, A. & Zipf, P. *An FPGA-optimized architecture of Horn and Schunck optical flow algorithm for real-time applications* in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)* (ieeexplore.ieee.org, 2014), 1. doi:10.1109/FPL.2014.6927406.

202. Ishii, I., Taniguchi, T., Yamamoto, K. & Takaki, T. High-Frame-Rate Optical Flow System. *IEEE Trans. Circuits Syst. Video Technol.* **22**, 105 (2012).

203. Wei, Z., Lee, D.-J. & Nelson, B. E. FPGA-based real-time optical flow algorithm design and implementation. *J. Multimedia* **2**. doi:10.4304/jmm.2.5.38-45 (2007).

204. Tanner, J. E. *Integrated optical motion detection* PhD thesis (California Institute of Technology, 1986).

205. Benson, R. G. & Delbrück, T. *Direction selective silicon retina that uses null inhibition* in *Advances in Neural Information Processing Systems 4 (NIPS 1991)* (eds Moody, J., Hanson, S. & Lippmann, R. P.) (Morgan-Kaufmann, Vancouver, 1992).

206. Delbruck, T. Silicon retina with correlation-based, velocity-tuned pixels. *IEEE Trans. Neural Netw.* **4**, 529 (1993).

207. Harrison, R. R. & Koch, C. A Robust Analog VLSI Motion Sensor Based on the Visual System of the Fly. *Auton. Robots* **7**, 211 (1999).

208. Liu, S.-C. A neuromorphic aVLSI model of global motion processing in the fly. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* **47**, 1458 (2000).

209. Lee, C., Kosta, A. K., Zhu, A. Z., Chaney, K., Daniilidis, K. & Roy, K. *Spike-flownet: event-based optical flow estimation with energy-efficient hybrid neural networks* in *European Conference on Computer Vision* (2020), 366. doi:10.1007/978-3-030-58526-6_22.

210. Zhu, A. Z., Yuan, L., Chaney, K. & Daniilidis, K. *Unsupervised event-based learning of optical flow, depth, and egomotion* in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2019), 989.

211. Pivezhandi, M., Jones, P. H. & Zambreno, J. *ParaHist: FPGA Implementation of Parallel Event-Based Histogram for Optical Flow Calculation* in *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)* (2020), 185. doi:10.1109/asap49362.2020.00038.

212. Xiong, W. & Chen, S. *U.S. pat. req.* 20210042939:A1 (2021).

213. Chen, S. *U.S. pat. req.* US20210037202A1 (2021).

214. Bierling, M. Displacement estimation by hierarchical blockmatching. *Visual Communications and Image Processing'88*. doi:10.1117/12.969046 (1988).

215. Li, J., Shi, F., Liu, W.-H., Zou, D., Wang, Q., Park, P. K. J. & Ryu, H. *Adaptive Temporal Pooling for Object Detection using Dynamic Vision Sensor* in *BMVC* (bmva.org, 2017). doi:10.5244/c.31.40.

216. Zhu, A. Z., Atanasov, N. & Daniilidis, K. *Event-based feature tracking with probabilistic data association* in *2017 IEEE International Conference on Robotics and Automation (ICRA)* (ieeexplore.ieee.org, 2017), 4465. doi:10.1109/ICRA.2017.7989517.

217. Rosten, E. & Drummond, T. *Machine learning for high-speed corner detection* in *European conference on computer vision* (2006), 430. doi:10.1109/icmlc.2014.7009151.

218. ARM. *AMBA 4 AXI4, AXI4-Lite, and AXI4-Stream Protocol Assertions User Guide, rop1* 2012.

219. Kuhn, P. *Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation* doi:10.1007/978-1-4757-4474-3 (Springer, Boston, MA, 1999).

220. Menze, M. & Geiger, A. *Object Scene Flow for Autonomous Vehicles* in *Conference on Computer Vision and Pattern Recognition (CVPR)* (2015). doi:10.1109/cvpr.2015.7298925.

221. Liu, S.-C., Rueckauer, B., Ceolini, E., Huber, A. & Delbruck, T. Event-Driven Sensing for Efficient Perception: Vision and Audition Algorithms. *IEEE Signal Process. Mag.* **36**, 29 (2019).

222. Sinangil, M. E., Sze, V., Zhou, M. & Chandrakasan, A. P. Cost and Coding Efficient Motion Estimation Design Considerations for High Efficiency Video Coding (HEVC) Standard. *IEEE J. Sel. Top. Signal Process.* **7**, 1017 (2013).

223. Chen, Y.-H., Krishna, T., Emer, J. S. & Sze, V. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE J. Solid-State Circuits* **52**, 127 (2017).

224. Aimar, A., Mostafa, H., Calabrese, E., Rios-Navarro, A., Tapiador-Morales, R., Lungu, I.-A., Milde, M. B., Corradi, F., Linares-Barranco, A., Liu, S.-C. & Delbruck, T. NullHop: A Flexible Convolutional Neural Network Accelerator Based on Sparse Representations of Feature Maps. *IEEE Trans Neural Netw Learn Syst* **30**, 644 (2019).

225. Sze, V., Chen, Y.-H., Yang, T.-J. & Emer, J. S. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* **105**, 2295 (2017).

226. Verhelst, M. & Moons, B. Embedded Deep Neural Network Processing: Algorithmic and Processor Techniques Bring Deep Learning to IoT and Edge Devices. *IEEE Solid-State Circuits Mag.* **9**, 55 (2017).

227. Delbruck, T. & Liu, S. *Data-Driven Neuromorphic DRAM-based CNN and RNN Accelerators* in *2019 53rd Asilomar Conference on Signals, Systems, and Computers* (2019), 500. doi:10.1109/IEEECONF44664.2019.9048865.

228. Richardson, I. E. *H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia* (John Wiley & Sons, 2004).

229. Komarek, T. & Pirsch, P. Array architectures for block matching algorithms. *IEEE Transactions on Circuits and Systems* **36**, 1301 (1989).

230. Brox, T. & Malik, J. Large displacement optical flow: descriptor matching in variational motion estimation. *IEEE transactions on pattern analysis and machine intelligence* **33**, 500 (2010).

231. Li, R., Shi, D., Zhang, Y., Li, R. & Wang, M. Asynchronous event feature generation and tracking based on gradient descriptor for event cameras. *International Journal of Advanced Robotic Systems* **18**, 17298814211027028 (2021).

232. Costante, G., Mancini, M., Valigi, P. & Ciarfuglia, T. A. Exploring representation learning with cnns for frame-to-frame ego-motion estimation. *IEEE robotics and automation letters* **1**, 18 (2015).

233. Muller, P. & Savakis, A. *Flowdometry: An optical flow and deep learning based approach to visual odometry* in *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)* (2017), 624.

234. Konda, K. R. & Memisevic, R. *Learning visual odometry with a convolutional network.* in *VISAPP (1)* (2015), 486.

235. Qin, T., Li, P. & Shen, S. Vins-mono: A robust and versatile monocular visual-inertial state estimator. *IEEE Transactions on Robotics* **34**, 1004 (2018).

236. Khairallah, M. Z., Bonardi, F., Roussel, D. & Bouchafa, S. PCA Event-Based Otical Flow for Visual Odometry. *arXiv preprint arXiv:2105.03760* (2021).

237. Gehrig, D., Loquercio, A., Derpanis, K. G. & Scaramuzza, D. *End-to-end learning of representations for asynchronous event-based data* in *Proceedings of the IEEE International Conference on Computer Vision* (2019), 5633.

238. Sironi, A., Brambilla, M., Bourdis, N., Lagorce, X. & Benosman, R. *HATS: Histograms of averaged time surfaces for robust event-based object classification* in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018), 1731.

239. Orchard, G., Jayawant, A., Cohen, G. K. & Thakor, N. Converting static image datasets to spiking neuromorphic datasets using saccades. *Frontiers in neuroscience* **9**, 437 (2015).

240. Xilinx, S. *PetaLinux Tools Documentation* 2018.

241. Kung, H.-T. Why systolic architectures? *Computer* **15**, 37 (1982).

242. Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., *et al. In-datacenter performance analysis of a tensor processing unit* in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (2017), 1.

243. De Fine Licht, J., Besta, M., Meierhans, S. & Hoefler, T. Transformations of high-level synthesis codes for high-performance computing. *IEEE Transactions on Parallel and Distributed Systems* **32**, 1014 (2020).

244. Cavigelli, L. & Benini, L. Extended Bit-Plane Compression for Convolutional Neural Network Accelerators. *arXiv preprint arXiv:1810.03979* (2018).

245. Specification, U. S. B. *USB 2.0 Specification* 2000.