Systematic Optimizations for Efficient Mobile Visual Computing

by

Yu Feng

Submitted in Partial Fulfillment of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Professor Yuhao Zhu

Department of Computer Science
Arts, Sciences and Engineering
Edmund A. Hajim School of Engineering and Applied Sciences

University of Rochester
Rochester, New York

2023

Dedicated to my loving parents, Qiong Feng and Qiuyun He, and my girlfriend, Kening Hu.

Table of Contents

Biographical Sketch

Prior to my time at the University of Rochester, I completed a Master's degree in Material Science from Carnegie Mellon University and a Bachelor's degree in Material Science from Tianjin Polytechnic University. During my undergraduate studies at Tianjin Polytechnic University, I was honored with a second-class scholarship twice and a third-class scholarship once. Additionally, I was awarded the Graduate Fellowship at Carnegie Mellon University.

In 2017, I started my journey at the University of Rochester, enrolling in the Ph.D. program within the Mechanical Engineering department. However, my research interests later led me to transition to the Computer Science department in 2018, under the guidance of Dr. Yuhao Zhu. Throughout my study at the University of Rochester, I also engaged in three internship experiences, including one with Google AI and two with Meta Reality Labs.

The following paragraph lists all works that are previously published in peer-reviewed conferences and journals:

- *PES: Proactive Event Scheduling for Responsive and Energy-Efficient Mobile Web Computing.* Yu Feng, and Yuhao Zhu. In Proceedings of the 46th International Symposium on Computer Architecture, 2019 (Feng and Zhu, 2019).

- *ASV: Accelerated Stereo Vision System.* Yu Feng, Paul Whatmough, and Yuhao Zhu. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019 (Feng et al., 2019).

- *Real-Time Spatio-Temporal LiDAR Point Cloud Compression.* Yu Feng, Shaoshan Liu, and Yuhao Zhu. In 2020 IEEE/RSJ international conference on intelligent robots and systems (IROS), 2020 (Feng et al., 2020a).

- *Mesorasi: Architecture Support for Point Cloud Analytics via Delayed-Aggregation.* Yu Feng, Boyuan Tian, Tiancheng Xu, Paul Whatmough, and Yuhao Zhu. In 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020 (Feng et al., 2020b).

- *Real-Time Gaze Tracking with Event-Driven Eye Segmentation.* Yu Feng, Nathan Goulding-Hotta, Asif Khan, Hans Reyserhove, and Yuhao Zhu. In 2022 IEEE Conference on Virtual Reality and 3D User Interfaces (VR), 2022 (Feng et al., 2022a).

- *Crescent: Taming Memory Irregularities for Accelerating Deep Point Cloud Analytics.* Yu Feng, Gunnar Hammonds, Yiming Gan, and Yuhao Zhu. In Proceedings of the 49th Annual International Symposium on Computer Architecture, 2022 (Feng et al., 2022b).

- *Fast and Accurate: Video Enhancement Using Sparse Depth.* Yu Feng, Patrick Hansen, Paul N. Whatmough, Guoyu Lu, and Yuhao Zhu. In Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision, 2023 (Feng et al., 2023).

- *CamJ: Enabling System-Level Energy Modeling and Architectural Exploration for In-Sensor Visual Computing.* Tianrui Ma, Yu Feng, Xuan Zhang, and Yuhao Zhu. In International Symposium on Computer Architecture (ISCA), 2023 (Ma et al., 2023).

Acknowledgments

First, I would like to express my heartfelt gratitude to Prof. Chen Ding, Prof. Anthony Rollett, Prof. Alan McGaughey, and Prof. Xiaoping Liang for their invaluable support throughout my academic journey. Their recommendation letters helped me gain admission to this esteemed department and provided me with the opportunity to pursue my passion for research. I am also thankful to this wonderful department for providing financial support during my Ph.D. journey.

I would like to express my deepest gratitude to my advisor, Yuhao Zhu, for his continuous guidance, support, and mentorship throughout my Ph.D. journey. I am grateful for his patience, encouragement, and dedication to helping me become a researcher. I learned so much from his vision and approach to research, I am fortunate to have had such a privilege of working under his supervision.

I would also like to thank the members of my dissertation committee, Prof. Sandhya Dwarkadas, Prof. Sreepathi Pai, Prof. Chenliang Xu, and Dr. Nathan Goulding-Hotta for their valuable feedback, which helped to develop the core ideas in my dissertation and refine my research work. Their expertise and encouragement have been instrumental in shaping my research and academic career.

Last but not least, I want to thank my family, especially my parents, Qiong Feng and Qiuyun He and my girlfriend, Kening Hu, for their love, support, and encouragement throughout my Ph.D. journey. Their support has driven me toward my success. I am grateful for their sacrifices. I would also like to thank my dedication to basketball and running, which have helped me build resilience during my research.

Abstract

Mobile visual computing is now a crucial facilitator for a range of applications such as autonomous vehicles and augmented/virtual reality (AR/VR), and we are continuously witnessing the development of new algorithms and devices that allow us to experience the world in novel ways. However, with the advent of the post-Moore's law era, the computing capabilities of current mobile devices are no longer sufficient to meet the increasing demands of computation required by today's applications. Moreover, high computation and data communication demands drain the device batteries quickly. Therefore, optimizing the visual computing stack holistically, from algorithm to hardware, is essential.

This dissertation makes a significant contribution by identifying bottlenecks in the two primary stages of the visual pipeline: sensing and processing. Particularly, this dissertation demonstrates how data communication is the major bottleneck in the sensing stage, while overwhelming computation demands hinder the performance of the processing stage. Based on such observations, this dissertation mitigates data communication overhead via two techniques: in-sensor computing and application-specific compression. Additionally, to address the computation bottlenecks, this dissertation exploits two application properties to avoid redundant computations: temporal correlation and algorithmic approximation. These techniques aim to optimize the entire visual computing stack, including hardware and algorithms, to achieve better energy-efficient and high-performance mobile vision systems.

List of Tables

List of Figures

Contributors and Funding Sources

This work was supervised by a dissertation committee consisting of Prof. Sandhya Dwarkadas, Prof. Sreepathi Pai, Prof. Chenliang Xu, Dr. Nathan Goulding-Hotta and Prof. Yuhao Zhu. This work includes collaborations with Paul Whatmough, Shaoshan Liu, Boyuan Tian, Tiancheng Xu, Patrick Hansen, Guoyu Lu, Nathan Goulding-Hotta, Asif Khan, Hans Reyserhove, Gunnar Hammonds, Yiming Gan, Tianrui Ma, Xuan Zhang, and Yuhao Zhu. This material is based upon work supported by the National Science Foundation Award 2044963 and a Meta research grant. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of above named organization.

1 Introduction

Mobile visual system is now a crucial facilitator in many vision applications, such as autonomous driving and AR/VR. Reports show that the market size of AR/VR is estimated to be 37.0 billion dollars in 2023 with an annual growth rate of 25.3% (Report, 2022). These technologies are continuously reshaping the way people communicate and interact with the world around them. Meanwhile, as we witness rapid innovations in technology, new emerging applications are invented with ever-increasing computation demands. While embracing the new experiences that are brought by these emerging applications, we also are facing a crucial fact that, in this post-Moore's law era, the general hardware performance is no longer expected to be doubled every two years by semiconductor technology developments. How to bridge the gap between the increasing computation demands from vision applications and the stagnated semiconductor technology is an urgent issue that needs to be addressed.

The challenge of rising demand and insufficient supply extends beyond just performance metrics, including energy and bandwidth. Such demand-supply mismatches impede the broader adoption of many vision applications. For instance, in applications such as high frame-rate and high dynamic range (HDR) digital image sensors, data capture rates are now constrained by the chip-to-chip or internet bandwidth rather than the analog-to-digital conversion, which is typically the bottleneck in conventional CMOS image sensors.

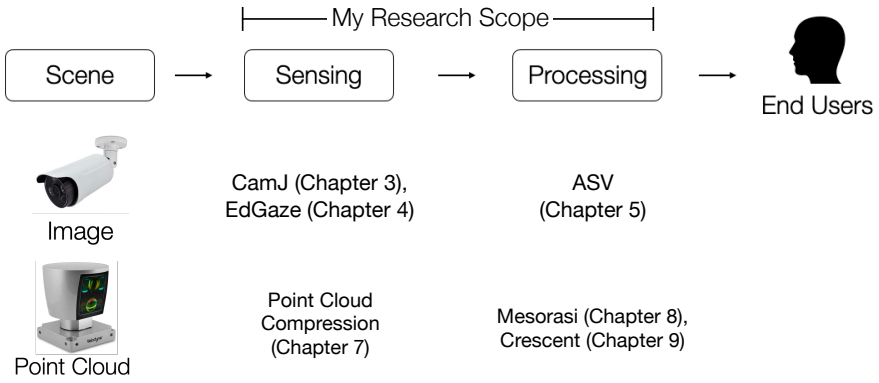


Fig. 1.1: Overview of my research scope.

The rest of this chapter is organized as follows. Sec. 1.1 outlines the research scope and contributions of this dissertation followed by a summary of the thesis statement in Sec. 1.2. Next, Sec. 1.3 highlights the long-term impacts of this dissertation. Sec. 1.4 overviews the organization of this dissertation and Sec. 1.5 lists the previous publications that this dissertation is based on.

1.1 Research Scope and Contributions

The general vision pipeline is illustrated in Fig. 1.1. At a high level, the vision pipeline starts with the scene, which is captured by the sensing devices that convert signals (e.g. lights) into electric signals. These converted signals then are processed in either the analog domain or digital domain before being presented to the end users or displays. This dissertation focuses on two key stages of the vision pipeline: sensing and processing. More specifically, it targets two main modalities in many mobile applications: image and point cloud. Based on the unique characteristics of these two modalities, this dissertation proposes different optimization techniques to address the fundamental inefficiencies in these two stages.

Under this research scope, my dissertation makes the following contributions.

In-Sensor Computing Algorithm and Framework The goal of in-sensor com-

puting is to resolve the data communication overhead in sensor readouts. However, I realize there are two essential pieces missing to allow sensor designers to fully utilize the power of in-sensor computing. The two missing pieces are a well-established algorithm paradigm and a design exploration framework. To this end, I first propose EDGAZE which set an example for an ideal in-sensor computing paradigm. Unlike existing vision algorithms which are infeasible to be deployed on today's image sensors, EDGAZE considers the computation limitations inside the image sensor and effectively reduces the overall energy consumption and data communication. To the infrastructure end, to better allow sensor designers or system architects to explore the sensor design space, I propose CAMJ which is the first-of-its-kind design exploration framework that allows quick system-level performance evaluation with a user-friendly interface.

Leverage Temporal Correlations in Computation Motions in videos encode the temporal correlations across frames. By exploiting this characteristic, I propose ASV which leverages the temporal correlations across image frames to avoid redundant computations in vision applications, therefore, reducing the overall computation cost. However, unlike many studies which simply use motions to approximate the next frame results, ASV leverages geometric properties in the depth estimation and carefully designs a domain-specific system to improve the overall performance.

Spatio-Temporal Point Cloud Compression Point clouds naturally encode spatial geometric information of an object or a scene. Exploiting this characteristic allows us to encode point clouds using geometric equations and effectively compress point clouds. Additionally, point cloud streams often exhibit temporal correlations similar to image-based videos. Using this feature, I propose a real-time point cloud compression algorithm. This algorithm leverages the characteristics of LiDAR-generated point clouds and exploits the geometric features of the physical world and temporal similarities across point clouds to encode point clouds.

Point Cloud Algorithm-Hardware Co-Designs Deep point cloud analytics gains popularity due to its power in many vision tasks such as classification and detection.

However, due to its unique computation operations, it is inefficient to execute deep point cloud algorithms on today's GPU or accelerators. This dissertation points out two fundamental inefficiencies in point cloud algorithms: inherently sequential execution and irregular memory access. To address the first inefficiency, I propose MESORASI, an algorithm-hardware co-design for deep point cloud analytics. MESORASI explores the approximate nature of deep learning networks and meticulously interchanges the order of the computation operands to achieve parallelism and reduce the overall computation. To address the second inefficiency, I propose CRESCENT, which approximates two key operations in point cloud analytics, kd-tree search and aggregation, and proposes corresponding hardware augmentation to further accelerate the point cloud algorithms. Compared to standard point cloud algorithms, CRESCENT reduces the irregular memory accesses to a large extent and saves the data communication energy.

1.2 Thesis Statement

To enable next-generation visual computing systems, we must improve the efficiency of both sensing and processing stages across different visual modalities. To this end, this dissertation addresses fundamental inefficiencies in both sensing and processing. To optimize sensing efficiency, this dissertation proposes novel in-sensor computing techniques and modality-tailored data compression methods to minimize the data communication overhead in vision pipelines. To optimize processing efficiency, an important observation made in this dissertation is that vision algorithms inherently involve approximation during the computation. This dissertation leverages such an observation to avoid exact computations through algorithm-hardware co-design, reducing the overall computations in vision pipelines.

1.3 Long-term Impact

The goal of this dissertation is to build real-time and energy-efficient mobile vision systems. This goal will be continuously influential as people are seeking more engaging visual experiences. The long-term impact of my work encompasses three aspects.

First, mobile vision systems remain critical components in many existing and emerging applications, including AR/VR, autonomous vehicles, robotics, etc. Many of those are considered the next-generation platforms and human-machine interfaces. More importantly, image and point cloud will continue to serve as two major visual modalities in many mobile vision systems. The techniques proposed in this dissertation will continue to benefit many future applications, particularly as data communication and computation remain the primary bottlenecks.

Second, as users require better user experience such as higher resolutions, higher frame rates, and smaller form factors, the resolution for mobile visual computing will inevitably increase with more constrained power/energy budgets. However, with Moore's law no longer holding, system architects must design more efficient mobile systems. The visions in this dissertation will continuously guide system designers to build vision platforms for future applications.

Lastly, in the post-Moore's law era, the hardware design inevitably shifts towards domain-specific design. However, the debate between general-purpose vs. domain-specific designs remains ongoing. The important message of this dissertation is that achieving the best system performance does not always mean designing a brand-new architecture. Instead, by co-designing hardware with algorithms, many mobile applications can largely avoid proposing unnecessary architectures and just exploit the existing architectures with minimum but principal hardware augmentation.

1.4 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 introduces the background of image sensors and image-based vision tasks. Chapter 3 presents CAMJ framework and shows how CAMJ can perform rapid system-level performance and power evaluation. Next, Chapter 4 explores the capability of CAMJ and quantitatively demonstrates how EDGAZE can leverage in-sensor computation power to reduce the overall data communication. Chapter 5 shifts the focus on the processing stage of the vision pipeline and describes how the systematic optimization in ASV is tailored for the fundamental inefficiencies in depth-estimation and avoids redundant computations in depth-estimation algorithms. Chapter 6 introduces the basic mechanism in point cloud acquisition and the basic operands in deep point cloud analytics. After that, Chapter 6 highlights some challenges in the sensing and processing stages in the point cloud domain. Based on the aforementioned challenges, Chapter 7 introduces a real-time point cloud algorithm to address the data communication challenge in point cloud sensing, and Chapter 8 and Chapter 9 address the compute and memory challenges in point cloud processing, respectively.

1.5 Published Materials

This dissertation contains materials that are previously published in peer-reviewed conferences and journals:

Chapter 3. The design and validation of in-sensor framework, CAMJ, are based on the following paper: *CamJ: Enabling System-Level Energy Modeling and Architectural Exploration for In-Sensor Visual Computing.* Ma, Tianrui, Yu Feng, Xuan Zhang, and Yuhao Zhu. In International Symposium on Computer Architecture (ISCA), 2023 (Ma et al., 2023).

Chapter 4. The algorithm design of EDGAZE is based on the following pa-

per: *Real-time gaze tracking with event-driven eye segmentation.* Feng, Yu, Nathan Goulding-Hotta, Asif Khan, Hans Reyserhove, and Yuhao Zhu. In 2022 IEEE Conference on Virtual Reality and 3D User Interfaces (VR), 2022 (Feng et al., 2022a). The in-sensor architecture design is based on the following paper: *CamJ: Enabling System-Level Energy Modeling and Architectural Exploration for In-Sensor Visual Computing.* Ma, Tianrui, Yu Feng, Xuan Zhang, and Yuhao Zhu. In International Symposium on Computer Architecture (ISCA), 2023 (Ma et al., 2023).

Chapter 5. The real-time depth estimation system, ASV, is based on the following paper: *Asv: Accelerated stereo vision system.* Feng, Yu, Paul Whatmough, and Yuhao Zhu. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019 (Feng et al., 2019).

Chapter 7. The real-time point cloud compression method is based on the following paper: *Real-time spatio-temporal lidar point cloud compression.* Feng, Yu, Shaoshan Liu, and Yuhao Zhu. In 2020 IEEE/RSJ international conference on intelligent robots and systems (IROS), 2020 (Feng et al., 2020a).

Chapter 8. The delayed-aggregation method and hardware augmentation are based on the following paper: *Mesorasi: Architecture support for point cloud analytics via delayed-aggregation.* Feng, Yu, Boyuan Tian, Tiancheng Xu, Paul Whatmough, and Yuhao Zhu. In 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020 (Feng et al., 2020b).

Chapter 9. The algorithm-hardware co-design system, CRESCENT, is based on the following paper: *Crescent: taming memory irregularities for accelerating deep point cloud analytics.* Feng, Yu, Gunnar Hammonds, Yiming Gan, and Yuhao Zhu. In Proceedings of the 49th Annual International Symposium on Computer Architecture, 2022 (Feng et al., 2022b).

2 Image-based Vision System

This chapter provides basic backgrounds about image-based vision systems. Specifically, Sec. 2.1 gives an overview of the CMOS image sensor and explains the role of an image sensor in today's vision pipelines. Next, Sec. 2.2 discusses the recent design trend in image sensors and showcases several sensor designs. Finally, Sec. 2.3 reviews common trade-offs when choosing different vision algorithms in today's mobile vision systems.

2.1 CMOS Image Sensor

This section provides the basic fundamentals of modern image sensors. This section starts with the role of an image sensor in general imaging or vision pipelines (Sec. 2.1.1). Then, the conventional architecture of an image sensor is used as an example to introduce the key components inside the image sensor (Sec. 2.1.2) and peripheral readout circuitry (Sec. 2.1.3) followed by a brief summary (Sec. 2.1.4). This section focuses more on the mechanism of each individual component (from a computer science perspective) instead of its actual circuit-level implementation.

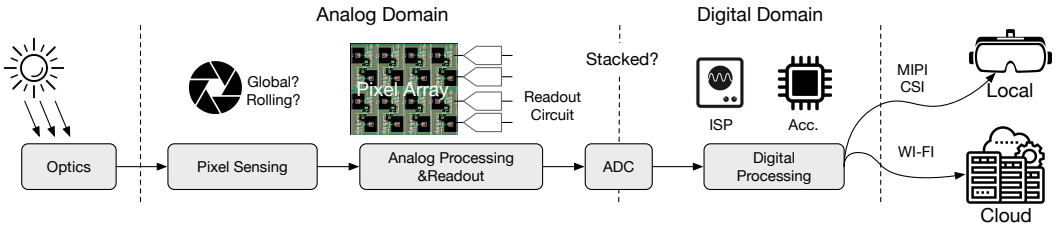


Fig. 2.1: An overview of the camera imaging and vision pipeline.

2.1.1 Overview

Fig. 2.1 shows an illustration of a general image-based vision pipeline. The vision pipeline starts with the sensor exposure, during which lights pass through optic components and hit pixels inside the image sensor. Photons received by pixels are then converted into currents via quantum effects. The converted currents are stored in the pixel well in the form of electrons. Once each pixel has accumulated enough electrons in one exposure, the accumulated electrons go through a series of analog signal processing and are eventually converted into voltage signals. Analog-to-digital converters (ADCs) receive voltage signals and digitize voltage signals so that the output signals can be processed by general digital processors, such as image signal processors (ISPs), or other domain-specific accelerators. Eventually, the processed digital signals are consumed by local mobile devices or remote clouds. In this section, we focus on one of the essential components in this pipeline, the image sensor.

2.1.2 Architecture of Computational Image Sensor

Although the designs of image sensors are continuously evolving in the literature, the fundamental principles of image sensors remain the same. To understand the fundamental architectural design of a computational image sensor (CIS), this section uses a 2D CIS design as an example. In principle, a CIS needs to fulfill two key requirements, image capturing and image processing. Fig. 2.2 shows a 2D CIS design consisting of a 2D pixel array that converts photons to analog signals, a 1D array of ADCs that con-

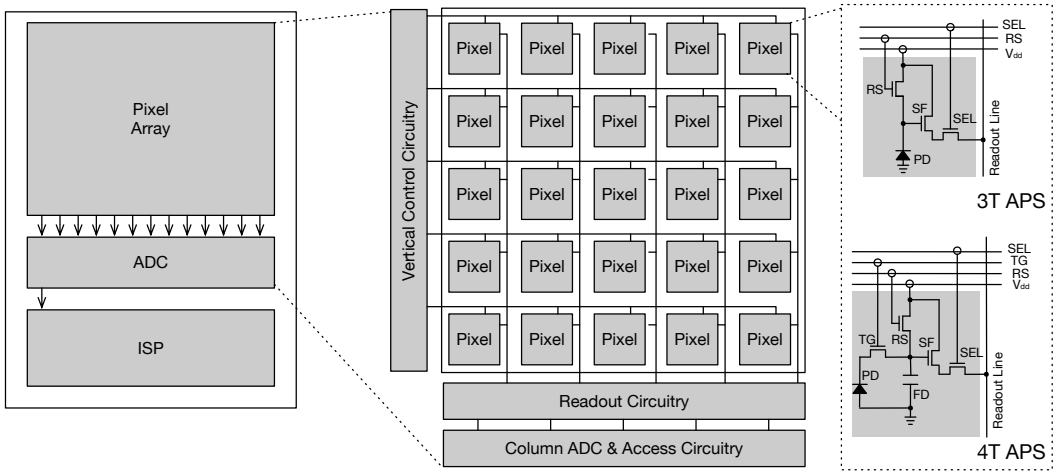


Fig. 2.2: An example of a 2D image sensor, which consists of a pixel array, a column ADC, and an ISP. Here, we highlight two different pixel designs, 3T-APS and 4T-APS.

verts the analog signals to digital values, and an ISP for image pre-processing, all in one single chip. This section focuses solely on the image-capturing aspect of CIS: pixel array and ADC. More details of ISP can be found in reference (Ramanath et al., 2005).

An example of a 2D pixel array is shown in Fig. 2.2. Each pixel array consists of a 2D array of pixels, along with some control circuitry and readout circuitry. Each pixel consists of a photodiode and some transistors to amplify, control, and transfer signals. Further details regarding these components are given in the following paragraphs. After being processed by these components, signals are read out by the readout circuitry in a row-by-row fashion. The order of readout is controlled by the vertical control circuitry. Only one row can be accessed at a time. The horizontal readout circuitry reads a row of pixel signals in parallel and performs some analog processing before sending the signals to the column ADC. The column ADC then converts a row of analog signals into digital signals (also in a parallel fashion) and outputs them to some digital registers or buffers for subsequent digital processing. The following paragraphs explain the functionality of each component inside the pixel in depth.

Photodiode The main functionality of a pixel is to sense light and convert its

intensity into a measurable form, namely an analog signal. During pixel capturing, lights initially hit the photodiode (PD), which converts photons to electrons via the quantum effect,

$$E(L) = P_{QE} \times P_{PD}, \quad \text{where} \quad P_{QE} \sim N(\mu = G_{QE}, \sigma = \sigma_{QE}) \quad \text{and} \quad P_{PD} \sim Poisson(\mu = \bar{L}, \sigma = \sqrt{\bar{L}}) \tag{2.1}$$

where P_{QE} is the quantum efficiency coefficient, which follows a normal distribution. G_{QE} is the expectation (the ideal quantum efficiency of the pixel), and σ_{QE} is the standard deviation across pixels. The actual number of photons that can hit the photodiode (P_{PD}) can be modeled as a Poisson distribution, denoted by $Poisson(\mu = \bar{L}, \sigma = \sqrt{\bar{L}})$, where $\bar{L}$ is the average number of electrons that hit the photodiode. The expectation of this Poisson distribution is $\bar{L}$ and the standard deviation of it is $\sqrt{\bar{L}}$). Such deviation from the expected value due to the uncertainty of the incident photons is known as shot noise. The converted electrons are then accumulated in the pixel well inside PD (Fig. 2.2).

The common pixel design includes 3T active pixel sensor (3T-APS) and 4T-APS in modern image sensors. Here, "T" stands for transistors, and the number in front of "T" stands for the number of transistors inside each pixel. Fig. 2.2 shows these two pixel designs. The operation procedures of these two pixels are described as follows.

3T-APS At the beginning of the 3T-APS exposure period, the reset transistor (RS) is turned on so that photodiode (PD) is reset to a reset voltage (V_{RST}) corresponding to zero pixel value. Next, the RS is switched off and PD starts to accumulate electrons during the exposure period. The accumulated electrons are stored in the PD junction capacitor, also known as the pixel well. As the electrons are accumulated in the PD junction capacitor, the potential of PD decreases. The difference between the reset voltage (V_{RST}) and decreased potential value (V_{PD}) eventually corresponds to the output pixel value. After pixel exposure, the selection transistor (SEL) is switched on to output the pixel potential (V_{PD}) for subsequent processing. During this readout pe-

riod, the source follower (SF) amplifies the signal and suppresses subsequent noises to achieve a better SNR. After V_{PD} is read out, SEL is then switched off.

4T-APS One significant issue with 3T-APS is the presence of k_BTC noise, which is generated from the voltage fluctuation across the PD junction capacitor. This voltage fluctuation causes the connected RS transistor to generate heat which introduces noises. Therefore, the k_BTC noise from the RS transistor is known as reset noise. To address reset noise, 4T-APS introduces additional elements inside the pixel: one transfer gate (TG) and an additional capacitor (FD). During the sensing process of 4T-APS, RS first resets the FD potential to V_{dd}. SEL is then switched on to output the reset potential (V_{RS}) along with reset noise. Next, SEL is switched off and 4T-APS starts its exposure. During its exposure, PD starts to accumulate photons. After exposure, TG is switched on to transfer electrons from PD to FD. Then, TG is switched off and SEL is switched on again to output the actually potential V_{PD}. Different from 3T-APS, 4T-APS outputs two voltage values in one exposure: reset voltage, V_{RS}, and the actual potential, V_{PD}. By subtracting V_{RS} from V_{PD}, reset noise can be effectively suppressed. To process two output signals in one exposure, a technique called correlated double sampling (CDS) which can perform two-value subtraction is used and will be introduced in Sec. 2.1.3.

Other noises In addition to shot noise and reset noise, the signal-to-noise ratio (SNR) of pixels is also affected by two other noises: dark current noise and read noises from varying sources. Dark current noise is the result of randomly generated electrons during the pixel exposure. Dark current noise increases along with sensor exposure time and sensor operation temperature. Although dark current noise is often low (less than the order of 10 electrons per second) at room temperature, dark current noise can become significant due to long exposure or low light conditions. Read noise is generated by reading out from components, such as FD, SF, etc. It can be modeled as a zero-mean normal distribution, $N(\mu = 0, \sigma = \sigma_{FD/SF})$, where $\sigma_{FD/SF}$ is the standard deviation of FD/SF noise.

2.1.3 CIS Readout Circuitry

This section introduces the readout circuitry around the pixel array. Before being converted into digital values by ADC, pixel signals need to go through several analog components. Using 4T-APS in Fig. 2.2 as an example, the output signal from a pixel needs to go through column amplification, CDS, and ADC. These three components are introduced in the following paragraphs.

Column Amplification The main functionality of the column amplifier is to amplify signals. Amplifying signals instead of directly processing signals can effectively suppress the noises introduced from subsequent stages, namely, CDS and ADC. Although the column amplifier suppresses noises in subsequent stages, the column amplifier itself also introduces some noises. There are two noises introduced by the column amplifier: read noise and fixed pattern noise due to spatial non-uniformity. Mathematically, these two noises can be modeled as:

$$V_{out} = G_{col} \times V_{in} + N_{col}, \quad \text{where} \quad G_{col} \sim N(\mu = \overline{G_{col}}, \sigma = \sigma_{col}) \quad \text{and} \quad N_{col} \sim N(\mu = 0, \sigma = \sigma_{col\_read}) \tag{2.2}$$

where G_{col} is column amplifier gain and modeled as a normal distribution, where $\overline{G_{col}}$ is the expectation of column amplifier gain, and σ_{col} is the standard deviation. V_{in} is the input signal and N_{col} is the read noise from the column amplifier and the read noise follows a zero-mean normal distribution, $N(\mu = 0, \sigma = \sigma_{col\_read})$.

Correlated Double Sampling (CDS) As mentioned in Sec. 2.1.2, CDS can effectively remove the reset noise. Although there are a few different designs for CDS, the fundamental mechanism of CDS is the same. The way that CDS suppresses reset noise is to first take two voltage outputs from the pixel array, one is the reset voltage (V_{RST}) and the other is the potential voltage (V_{PD}). CDS then calculates the difference between these two values. Because both measurements are affected by reset noise, CDS can ef-

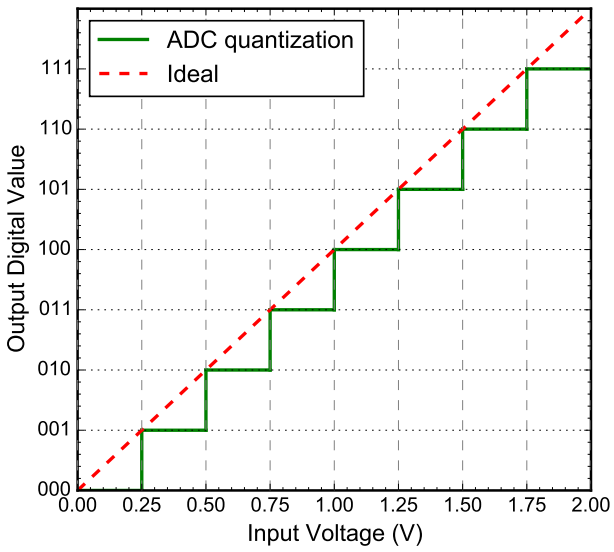


Fig. 2.3: An Example of 3-bit ADC quantization.

fectively suppress the reset noise. Although CDS itself also introduces some noises, CDS noise is quite small and can be ignored during the noise modeling.

Analog-to-Digital Converter (ADC) ADC is the interface between the continuous analog signal captured by pixel and the discrete digital signal consumed by digital processors. Various ADC designs exist in the literature, such as single-slope ADC, dual-slope ADC, successive-approximation ADC, sigma-delta ADC, etc. Each has its own performance-precision trade-off. For instance, successive-approximation ADC has a good balance between performance-precision and is one of the commonly used ADCs inside image sensors, while sigma-delta ADC can achieve much higher precision and is typically used for scientific measurements. (Jose and Del Rio, 2013; Tang et al., 2022) Rather than discussing the different types of ADCs, this dissertation focuses on the mechanism of ADC and discusses how to model an ADC inside the image sensor.

One of the key differences between analog signals and digital signals is that analog signals are continuous while digital signals are discrete. In order to use one digital value to represent a range of analog values, ADC design has to follow a quantization function.

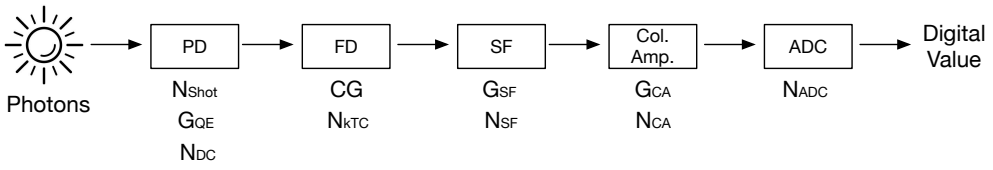


Fig. 2.4: Overall noise model. Here, we summarize all potential noise sources starting from the photon collection to the ADC readout.

Fig. 2.3 shows an example of such a 3-bit ADC quantization function compared against an ideal linear mapping. In Fig. 2.3, the input voltage is evenly split into 8 intervals, ranging from 0 to 2V. Any input values that fall into one particular interval, e.g. [0, 0.25), will be mapped to one single digital value, 000. As the input value is increased by 0.25V, the output value is increased by one least significant bit (LSB), which is 001, in this case. In this example, the output digital signal of the ADC is proportional to the input signal, so it is also called linear ADC. Other non-linear ADCs are proposed for varying purposes (Jonsson, 2010). However, since a range of input signals is mapped to a single output signal, inherently, ADC introduces quantization noise. The quantization noise is $LSB^2/12$, where LSB is:

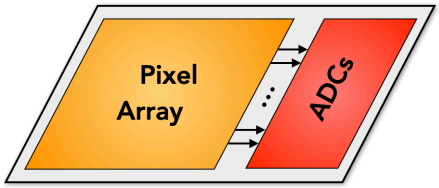
$$LSB = \frac{V_{max}}{2^{\#bit} - 1} \tag{2.3}$$

where $\#bit$, in this case, is 3, and V_{max} is the full scale input voltage 2V.

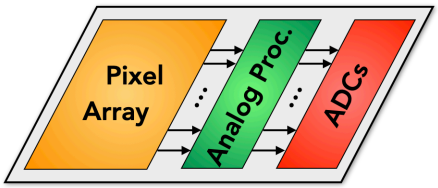
In reality, ADC usually deviates from the ideal ADC conversion function. There are linear and non-linear errors from actual ADC. Many ADC noises can be calibrated during the manufacture. For those noises that cannot be calibrated, a zero-mean normal distribution is used to model the overall ADC noise, $P_{ADC} \sim N(\mu = 0, \sigma = \sigma_{ADC})$.

2.1.4 Summary

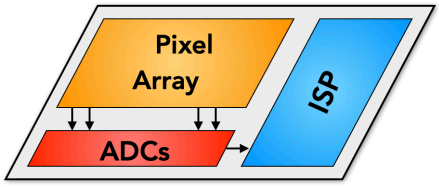
Fig. 2.4 summarizes the key stages in the image sensing pipeline with their associated noises. Starting with the photons incident the photodiode (PD), the noises include



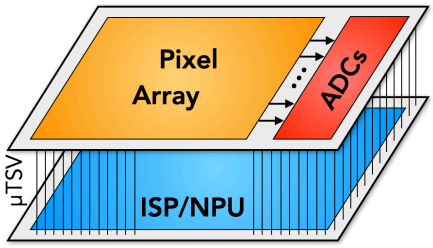
(a) Traditional 2D imaging CIS with photodiode array and ADCs.



(b) Computational CIS with analog processing capabilities.



(c) Computational CIS with digital accelerator (ISP here).



(d) Stacked computational CIS with digital accelerators in a separate layer.

Fig. 2.5: CIS architecture evolution. CIS is moving away from a purely imaging device (a) to integrate computation capabilities (b)(c), sometimes in a 3D stacking fashion (d).

shot noise (N_{Shot}), quantum efficiency non-uniformity (G_{QE}), and dark current noise (N_{DC}). Then, the floating diffusion (FD) converts electrons into voltages, and the associated noises with FD are non-uniformity of conversion gain (G_{CG}) and reset noise (N_{kTC}). Next, the source follower (SF) amplifies the signal with its own non-uniformity (G_{SF}) and its read noise (N_{SF}). During readout, the column amplifier amplifies signals again. The column amplifier also has its own gain non-uniformity (G_{CA}) and read noise (N_{CA}). Finally, the analog-to-digital converter (ADC) converts analog signals to digital signals with its own quantization error and ADC noise (N_{ADC}).

2.2 Design Trend of CMOS Image Sensor

Throughout the decades, image sensor design has undergone several major breakthroughs. In the early 1960s, the first concept of CMOS image has been proposed (Mor-

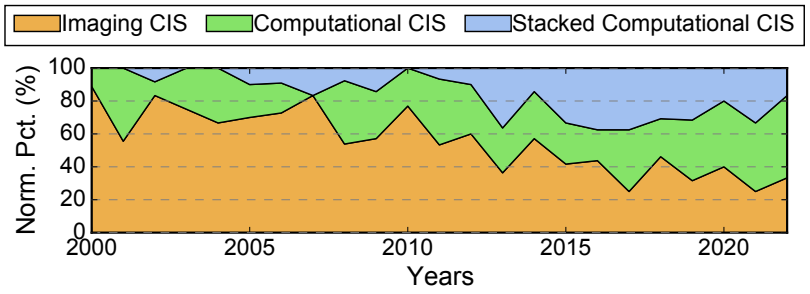


Fig. 2.6: Percentage of conventional CIS, computational CIS, and stacked computational CIS designs from surveying all ISSCC and IEDM papers published between Year 2000 and 2022.

rison, 1963; Horton et al., 1964). However, it was not until 1990 that CMOS image sensors were widely adopted by major sensor manufacturers due to the advancement of CMOS technology and started to replace the conventional charge-coupled device (CCD) technology. This major technology shift made image sensors more cost-effective and energy-efficient. In the late 1980s, the concept of the computational image sensor (CIS) was introduced (Bernard et al., 1993; Koch and Li, 1994). Image sensors were no longer passive-capturing devices and were given some basic processing capabilities to improve image quality. With the continuous development of silicon technology and 3D stacking, there is a growing trend to integrate more computational capabilities inside image sensors. To date, there is a variety of sensor designs in the literature. The following paragraphs introduce a few typical examples of image sensor designs.

Fundamentally, a CIS consists of two basic components as illustrated in Fig. 2.5a: a light-sensitive photodiode array that converts photons to charges and a read-out circuit that converts charges to digital values (i.e., raw pixels) through the analog-to-digital converters (ADC). Traditionally, raw pixels are transferred to the host, e.g., a Systems-on-a-Chip (SoC) on a smartphone, through the MIPI CSI-2 interface (Group, 2021). The Image Signal Processor (ISP) in the SoC removes sensing artifacts (e.g., denoising)

and prepares pixels for computer vision tasks and/or for visual display.

CIS Design Trend. A clear trend in CIS design is to move into the sensor computations that are traditionally carried out outside the sensor, which gives rise to the notion of *Computational CIS*. Fig. 2.6 shows the percentage of computational CIS papers in ISSCC and IEDM from Year 2000 and Year 2022 with respect to all the CIS papers during the same time range. Increasingly more CIS designs integrate compute capabilities.

The computations inside a CIS could take place in both the analog and the digital domain. Fig. 2.5b illustrates one example where analog computing is integrated into a CIS chip. Analog operations usually implement primitives for feature extraction (Bong et al., 2017b,a), object detection (Young et al., 2019), and DNN inference (Hsu et al., 2020; Xu et al., 2021). Fig. 2.5c illustrates another example that integrates digital processing, such as ISP (Murakami et al., 2022), image filtering (Kim et al., 2005) and DNN (Bong et al., 2017a).

As the processing capabilities become more complex, CIS design has embraced 3D stacking technologies, as is evident by the increasing number of stacked CIS in Fig. 2.6. Fig. 2.5d illustrates a typical stacked design, where the processing logic is separated from, and stacked with, the pixel array layer. The different layers communicate through hybrid bond or micro Through-Silicon Via (μTSV) (Tsugawa et al., 2017; Liu et al., 2022). The processing layer typically integrates digital processors; such as ISP (Kwon et al., 2020), image processing (Kumagai et al., 2018; Hirata et al., 2021), and DNN accelerator (LiKamWa et al., 2016; Eki et al., 2021). Three-layer stacked designs have been proposed. Sony IMX 400 (Haruta et al., 2017) integrates a pixel array layer, a DRAM layer, and a digital layer with an ISP. Meta conceptualizes a three-layer design (Liu et al., 2022) with a pixel array layer, a per-pixel ADC layer, and a digital processing layer that integrates a DNN accelerator.

Recent trends also show that the research focus has shifted from conventional 2D image sensors toward more complex 3D CIS designs. Fig. 2.6 shows the percentage

of different CIS designs from surveying all papers from ISSCC and IEDM between 2000 and 2022. In early 2000, the major research focus is still conventional 2D CMOS sensors. After 2010, more research groups start to look into CIS and put more focus on integrating 3D stacking technology in sensor designs.

It is no surprise that computational CISs emerge when energy efficiency is critical. From an architectural perspective, computational CIS offers two main energy benefits.

First, moving computation inside the sensor allows the pixel data to be consumed closer to where they are generated. Doing so reduces the data transmission energy, which could dominate the overall energy consumption. Specifically, data communication inside a CIS using a μTSV consumes about 1 pJ/B, whereas the energy cost of transmitting one Byte out of the CIS through the MIPI CSI-2 interface consumes about 100 pJ of energy (Liu et al., 2022). As an example, if a CIS is capable of executing an object detection DNN directly, the data volume that has to be transmitted out of the sensor is simply a few Bytes, as opposed to, say, 6 MB, for a 1080p image.

Second, computational CIS also provides a natural platform for analog acceleration, since the pixel data originate from the analog domain to begin with, obviating the need for energy-intensive digital-to-analog converters that often dominate the hardware overheads in conventional analog accelerators. Compared to digital processing, analog processing minimizes energy-intensive data conversion (Cao et al., 2021; Ma et al., 2022) and can reduce both computation and memory energy consumption.

2.3 Trade-offs in Image-based Vision Algorithms

The demand for intelligent applications running on a diverse range of mobile and embedded platforms, such as micro-robots, augmented reality headsets, and smart-city sensor nodes, shows no sign of slowing down. Image, being the primary modality in vision tasks, has played a crucial role in many of these intelligent applications.

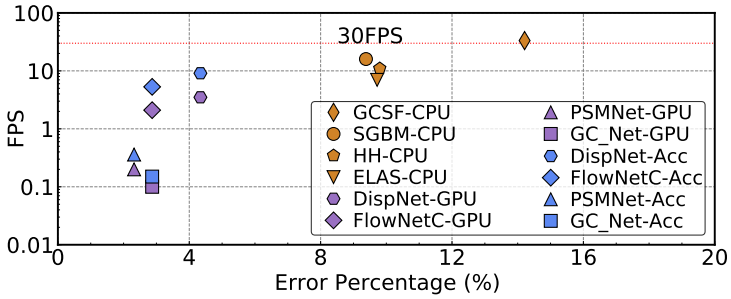


Fig. 2.7: demonstrates both real-time (30 FPS) frame rates and DNN-like accuracy for stereo vision.

Over the years, image-based vision algorithms have evolved rapidly. Prior to 2015, the majority of vision algorithms still reply on "hand-crafted" feature descriptors such as SIFT (Ng and Henikoff, 2003), HOG (Dalal and Triggs, 2005), etc. However, as deep learning has completely taken off in 2015 (LeCun et al., 2015), learned feature descriptors have been proposed to replace "hand-crafted" features with much higher accuracy and robustness. Nevertheless, both classic algorithms and deep neural network (DNN) algorithms presented to date broadly define a frontier in the accuracy-efficiency design space. Fig. 2.7 present such an accuracy-efficiency tradeoff in one of the key vision tasks, stereo depth estimation. Fig. 2.7 compares the frame rate and accuracy for four well-known classic stereo algorithms that use "hand-crafted" features, including GCSF (Cech et al., 2011), SGBN (Hirschmuller, 2005), HH (Hirschmuller, 2005), and ELAS (Geiger et al., 2010), as well as four state-of-the-art DNNs solutions (Mayer et al., 2016; Kendall et al., 2017; Chang and Chen, 2018; Smolyanskiy et al., 2018). The DNN data is characterized on both a Pascal mobile GPU (Nvidia, 2017c) ("-GPU" suffix), as well as on a DNN accelerator (Samajdar et al., 2018) ("-Acc" suffix). In using low-dimensional "hand-crafted" features, classic algorithms lead to high error rates (x-axis), but are computationally efficient, mostly operating at close to real-time (e.g., 30 FPS, y-axis). In contrast, DNNs models achieve very low error rates, but require 2–5 orders of magnitude more arithmetic operations, resulting in much lower frame rates.

The accuracy-performance trade-offs observed in stereo depth estimation are not unique, as similar trade-offs can be found in other computer vision tasks, such as object detection (Zhu et al., 2018b). Such accuracy-performance trade-offs impose interesting research questions when it comes to designing and integrating vision system stacks. How to balance accuracy and performance for a particular computing platform will become an ongoing research question to explore. It also requires collaborative work from both the computer vision community and the computer architecture community.

3 CamJ: In-Sensor Computing Exploration Framework

Studies show that the energy consumption of data transmission per byte is often orders of magnitude higher than that of computation. (Liu et al., 2022) Significant energy consumption on data communication poses an important challenge to battery life. Meanwhile, with the continuous development of CMOS technology, today's image sensor is no longer a passive capturing device. Sensors nowadays offer *it-situ* processing capabilities, ranging from light-weighted signal processing (Bong et al., 2017b,a) to moderate DNN inference (Hsu et al., 2020; Xu et al., 2021). This technology trend poses a question to mobile architects and system designers: how to leverage this limited computation capacity inside sensors to improve the overall system energy efficiency? The seemingly obvious solution is to use the in-sensor compute capability to preprocess the captured images so that less data need to be transmitted back to the SoC. While the idea is simple, there are many challenges to making in-sensor computing a reality.

To address these challenges, this chapter proposes an in-sensor design exploration framework called CAMJ which allows users to freely explore different CIS design choices. Specifically, this chapter begins by addressing the key challenges associated with in-sensor computing in Sec. 3.1. Sec. 3.2 introduces CAMJ framework. Sec. 3.3 discusses how CAMJ models the performance of different image sensors and provides detailed validation in Sec. 3.4. Finally, Sec. 3.5 discussed the related work.

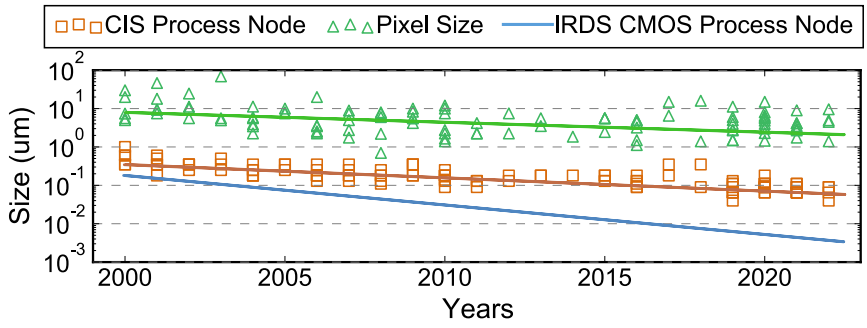


Fig. 3.1: CIS process node always lags behind conventional CMOS process node. This is because CIS node scaling tracks the pixel size scaling, which does not shrink aggressively due to the fundamental need for maintaining photon sensitivity.

3.1 Challenges in In-Sensor Computing

Moving computation inside a CIS, however, is not without challenges. Most importantly, processing inside the sensor is far less efficient than that outside the sensor, fundamentally because the CIS process node significantly lags behind that of the conventional CMOS. Fig. 3.1 illustrates this difference, where square markers show the process nodes used in CIS designs from all ISSCC papers appearing during 2000 to 2022, which include leading industry CIS designs at different times. We overlay a trend line regressed from these CIS designs to better illustrate the scaling trend. As a comparison, the blue line at the bottom represents the conventional CMOS technology node scaling laid out by International Roadmap for Devices and Systems (IRDS) (IRDS, 2022).

Around the year 2000, the CIS process node started lagging behind that of the conventional CMOS node, and the gap is increasing. CIS design today commonly use 65nm and older process nodes. This gap is not an artifact of the CIS designs we pick; it is fundamental: there is simply no need to aggressively scale down the process node because the pixel size does not shrink much. The triangles in Fig. 3.1 represent the pixel sizes of all the CIS designs we surveyed. The slope of CIS process node scaling

almost follows exactly that of the pixel size scaling. The reason that pixel size does not shrink is to ensure light sensitivity: a small pixel reduces the number of photons it can collect, which directly reduces the dynamic range and the Signal-to-Noise ratio (SNR) (Bigas et al., 2006).

Inefficient in-sensor processing can be mitigated through 3D stacking technologies (Xie and Zhao, 2015), which allows for heterogeneous integration: the pixel layer and the computing layer(s) can use their respective, optimal process node. In this way, while the pixel layer still (necessarily) has to use older process nodes, the computing logic could use advanced CMOS nodes similar to those used outside the CIS, minimizing the energy overhead of computing inside CIS. Stacking, however, could increase power density especially when future CIS integrate more processing capabilities. Therefore, harnessing the power of (stacked) CIS requires exploring a large design space and addressing key challenges, some of which we list below.

- Whether and what to compute in vs. off CIS?
- What to compute in the analog vs. digital domains?
- How to architect each layer in stacked CIS to achieve energy reduction without increasing power density?

3.2 CamJ Framework

No framework to date allows designers to explore the complicated design space of computational CIS at a system level. Our CamJ framework is designed to fill this void (Sec. 3.2.1). We first outline the design principles of CamJ, followed by an overview of the CamJ design internals (Sec. 3.2.2). We use a concrete example to demonstrate from a designer perspective how CamJ is used (Sec. 3.2.3).

3.2.1 When is CAMJ Used in the Design Cycle?

CAMJ is meant to be used for *system-level* exploration after each component design is sketched out; an analogy would be Systems-on-a-Chip (SoC) vs. accelerator design. Before system-level exploration, a team usually has at hand a range of component-level designs, which could be licensed Intellectual Property (IP) blocks, reference designs from the literature, or earlier designs from other teams in the organization (e.g., using a synthesis flow or High-Level Synthesis tools); in all cases the component-level energy behavior is known or can be modeled using external tools like Aladdin (Shao et al., 2014) and OpenRAM (Guthaus et al., 2016).

CAMJ helps designers make design decisions when assembling the individual (digital and analog) components into an optimal system. Ideally, a designer uses CAMJ to estimate the system energy given initial designs of individual components; using the estimation, a designer can *iteratively* refine the components/system design. For instance, CAMJ can identify energy bottlenecks and guide the re-design of corresponding components. Orthogonally, a designer can use CAMJ to explore optimal mapping and partitioning of the algorithms between analog vs. digital domains or in vs. off CIS to minimize overall system energy under performance targets.

CAMJ is *not* a synthesis tool; it does not generate (nor estimate the energy of) an accelerator. Rather, CAMJ can be used in conjunction with HLS: one could use HLS to first generate an accelerator and then use CAMJ to explore, in the bigger system, how/whether that accelerator would fit in a computational CIS to maximize end-to-end application gains.

3.2.2 Design Principles and Overview

As with any energy modeling tool (Brooks et al., 2000; Leng et al., 2013; Kandiah et al., 2021), the total CIS energy is sum of the product of 1) the access count to each hardware unit and 2) the per-access energy consumption. Therefore, the central ob-

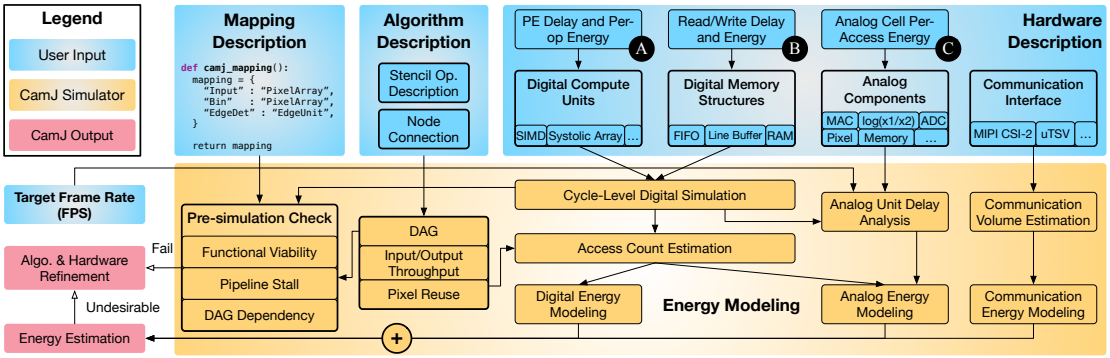


Fig. 3.2: Overview of the CAMJ framework, which provides a component-level energy estimation under a target frame rate (FPS). Users provide an algorithm and hardware description and the mapping between the two. Ⓐ, Ⓑ, and Ⓒ denote the per-access energy of digital computation, digital memory, and analog units, respectively; the first two are obtained from external tools while the last one is provided within CAMJ. See Tbl. 3.1 for a list of digital compute/memory and analog components supported in CAMJ. CAMJ then estimates the access count to each hardware component to obtain the overall energy estimation.

jective of CAMJ is to develop a modeling methodology that accurately estimates those two statistics using a programming interface that, critically, only requires user inputs for information that cannot be automatically inferred. Fig. 3.2 shows an overview of the CAMJ framework. Sec. 3.2.3 and Sec. 3.3 discuss the programming interface and modeling details, respectively. Here, we provide an overview of the interface and modeling methodology.

Interface. CAMJ observes that CIS deal with stencil-based image processing, which has regular computation and memory access patterns (i.e., little to no control flow) that are statically mapped to hardware units. The access count statistics can, thus, be inferred with only the high-level algorithm description and hardware configuration without knowing the implementation details. Therefore, CAMJ exposes a *declarative* interface, in which an algorithm is described by a DAG and the input/output/stencil dimensions at each node and the hardware is abstracted as a set of basic units, each

performing a specific sensing, computation, or memory operation.

CAMJ's interface also *decouples* the description of an algorithm, the underlying hardware, and the mapping between the two. A decoupled interface facilitates an iterative system design process, during which algorithm, hardware, and algorithm to hardware mapping can change independently. For instance, one can evaluate algorithmic changes by re-writing the algorithm description without touching the hardware design, or explore different algorithm-to-hardware mappings (e.g., split between analog vs. digital and between in vs. off sensor) by describing a new mapping.

Internal Modeling. CAMJ judiciously uses different methodologies to obtain the per access energy in the digital vs. analog domains. For digital structures, CAMJ directly asks users to provide the per-cycle energy of computation PEs (Ⓐ) and the per-access energy of memory units (Ⓑ). These statistics are usually obtained by an ASIC synthesis flow or from commonly used tools (e.g., CACTI (Balasubramonian et al., 2017) and OpenRAM (Guthaus et al., 2016)), and are routinely used in today's digital accelerator simulators for energy estimation (Gao et al., 2017; Akhlaghi et al., 2018; Kodukula et al., 2021b).

One might be surprised to find that CAMJ directly asks for the per-cycle/access energy of the digital structures. This is because of the design philosophy of CAMJ (Sec. 3.2.1): it is *not* used to generate digital accelerators; rather, it helps assess how an accelerator fits in the entire computational CIS system. For that reason, CAMJ expects designers to have a preliminary design of the digital accelerators (whether it's a manual design, HLS generated, or a licensed IP), in which case one will have the per-cycle/access energy statistics.

Unlike digital structures, few energy modeling tool exists for analog structures, whose energy consumption (Ⓒ) depend on many low-level circuit details (e.g., load capacitance, gain, bias current) that are cumbersome and perhaps unreasonable to ask system-level designers for. Our design decision is to expose a low-level interface to accept these parameters from expert users, but also provide default energy models based

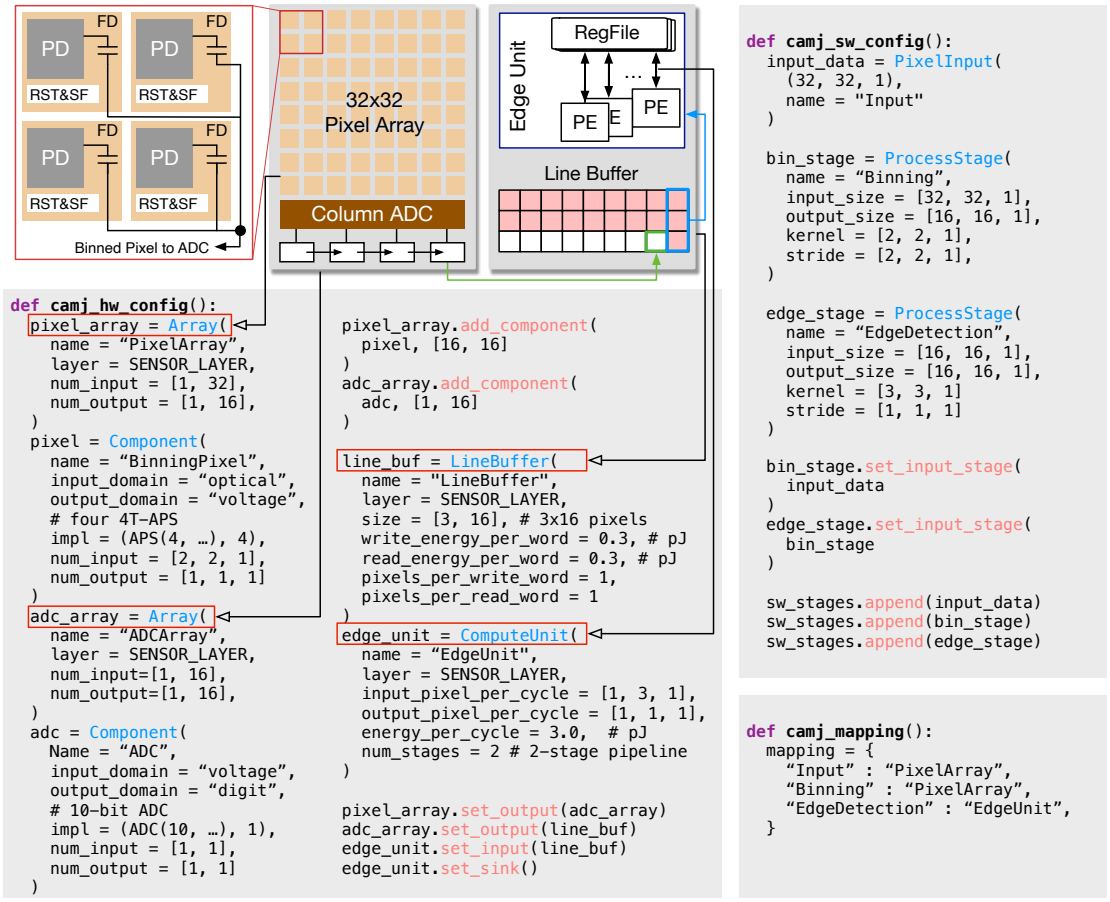


Fig. 3.3: An example of defining a simple CIS using the CAMJ programming interface in Python. The hardware architecture of the simulated CIS is illustrated at the top.

on classic implementations of analog components and delay analyses of these components (Sec. 3.3.1).

Finally, CAMJ performs a series of pre-simulation design checks to ensure that the algorithm and hardware combination 1) is functionally viable (e.g., ADCs must exist between the analog and digital domain), 2) does not have pipeline stalls (to avoid accumulating long frame latency), and 3) has well-formed dependencies in the algorithm DAG. We provide feedback upon check failures and a detailed energy breakdown, which helps designers iteratively refine the algorithm and/or hardware.

3.2.3 Programming Interface

We use a running example in Fig. 3.3 to introduce the programming interface and illustrate the main design decisions.

An Example. The Python code in Fig. 3.3 shows a concrete example to use the programming interface of CAMJ. In this conceptual CIS design with a 32×32 pixel array, every 2×2 pixel tile is first averaged (i.e., "binned") to produce a 16×16 image. The sensor then performs a digital edge detection on the image before sending the edge data through the MIPI CSI interface. The `camj_hw_config` function and the `camj_sw_config` function describe the hardware components and the algorithm, respectively. The `camj_mapping` maps each algorithm stage to a hardware component. We explain each part next.

Algorithm Description. The code in `camj_sw_config` describes the DAG of the entire processing pipeline, starting from the raw pixels generated by the pixel array (`PixelInput`), which go through two processing stages: `bin_stage` for pixel binning and `edge_stage` for edge detection. The `set_input_stage` method connects the stages together to form a DAG.

Notice how the algorithm description does not require the actual arithmetic details; we observe that image processing algorithms can be abstracted as stencil operations that operate on a local window of pixels at a time (Qadeer et al., 2013; Hegarty et al., 2014) — convolution (or image filtering in conventional image processing parlance) being a prime example. This observation holds in all the ISSCC/IEDM papers since Year 2000 we surveyed. Irregular computations complicate hardware design and increase energy, defeating the purpose of in-CIS computing.

Therefore, users express only the input/output image dimensions (`input_size`, `output_size`) along with the stencil window (`kernel`) and stride size (`stride`). Given the regular computation and data access pattern of stencil operations, CAMJ could accurately estimate the access counts to different hardware structures for energy

Table 3.1: A list of hardware units supported in CAMJ. APS: Active Pixel Sensor; DPS: Digital Pixel Sensor; PWM: Pulse Width Modulation; MAC: Multiply-Accumulate.

	Analog (A-COMPONENT)	**Digital**
Memory	Passive/Active, Sample-and-Hold	FIFO, Line Buffer, Double-Buffered SRAM
Compute	Pixel (APS, DPS, PWM), ADC, MAC, Max, Scaling, Add, log, Abs, Comparator	Systolic Array, Generic Pipelined Accelerator

estimations. Nonetheless, CAMJ does accept as input a memory trace offline collected for an irregular algorithm, which can then be integrated with external tools such as DRAMPower (Chandrasekar et al., 2012) to estimate the energy consumption for irregular algorithms.

Hardware Description. `camj_hw_config` describes the hardware architecture, which we illustrate at the top of Fig. 3.3. The hardware description consists of two components: analog processing units and digital processing units.

❶ Analog Units. CIS hardware necessarily starts from analog units, which, at a high level, are described as a set of Analog Functional Arrays (AFA), which is in turn is composed of a set of Analog Functional Components (A-COMPONENTs). The most important AFA in a CIS is the pixel array (`pixel_array`), in which each A-COMPONENT is a pixel, which is added to the pixel array through the `add_component` method. In the example of Fig. 3.3, the pixel array is followed by another AFA, i.e., the ADC array (`adc_array`), where each A-COMPONENT is an ADC.

From users' perspective, each A-COMPONENT performs a particular kind of (arithmetic) operation. In addition to a pixel or an ADC, CAMJ provides other common A-COMPONENTs used in CIS such as MAC or logarithmic operations. The complete

list of analog A-COMPONENTs is in Tbl. 3.1. The energy consumption of each A-COMPONENT, which is dictated by its circuit-level implementation, is abstracted away from the users by the `impl` method. Sec. 3.3.2 will later describe how we model the energy of each A-COMPONENT by mapping it to its analog circuit implementation.

What users do have to provide, however, is the signal dimension (`num_input` and `num_output`) and signal domain (`input_domain` and `output_domain`) of an AFA's input and output data. These parameters allow CAMJ to check whether the simulated CIS is functionally viable. Specifically, the `input_domain` of a consumer unit and the `output_domain` of a producer unit must match. If, for instance, the producer is in the charge domain and the consumer is in the voltage domain, CAMJ will ask designers to insert a charge-to-voltage conversion component[1], which has energy implications. Similarly, if the `num_input` of a consumer unit and the `num_output` of a producer unit do not match, the hardware must have an analog buffer in-between, which, again, could have energy implications.

❷ Digital Units. The digital part of the hardware is described by specifying a set of compute units that communicate through memory structures. In this example, the compute unit is the edge detection accelerator (instantiated through `ComputeUnit`), which reads from the line buffer (`LineBuffer`), a pre-defined memory structure, that stores data from the pixel array, an analog unit as described before.

Column 2 of Tbl. 3.1 lists the memory structures and compute units available in CAMJ. We support three memory structures commonly found in image/vision processing: FIFO (`FIFO`), line buffer (Hegarty et al., 2014; Whatmough et al., 2019b) (`LineBuffer`), and double-buffered SRAM (`DoubleBuffer`). The compute units are abstracted as pipelined accelerators through the `ComputeUnit` interface. We also provide a `SystolicArray` class to describe a systolic array due to its importance in executing DNNs.

[1] unless the output of the consumer is in the voltage domain, where the inherent capacitor of the consumer naturally acts as an analog buffer.

With the generic pipelined accelerator interface (`ComputeUnit`), users can model a wide range of (image processing) accelerators. To describe a pipelined accelerator, CamJ requires three main parameters: the shape of pixels read per cycle (`input_pixel_per_cycle`), the shape of pixels generated per cycle (`output_pixel_per_cycle`), and the pipeline depth (`num_stages`). Using these statistics, CAMJ performs cycle-level simulation for two purposes. First, CAMJ can check whether the accelerator will stall the CIS pipeline and, if so, asks for a re-design of the accelerator. Second, CAMJ can estimate the total latency of the digital domain, which is critical for analog energy estimation. Stall checking and latency estimation are critical for analog energy estimation as we will discuss in Sec. 3.3.1.

Mapping. The `camj_mapping` function maps each algorithm stage to a hardware unit. The code is self-evident. Users can simply remap the algorithm to hardware to explore a different system design. The decoupling of algorithm and hardware description through the mapping function also enables easy expression of hardware reuse—by simply mapping different algorithm nodes to a hardware component.

3.3 Energy Modeling Methodology

The energy consumption per frame of a CIS sensor is the sum of that of the analog, digital, and data communication:

$$E^{\text{frame}} = E_{\text{a}}^{\text{frame}} + E_{\text{d}}^{\text{frame}} + E_{\text{c}}^{\text{frame}} \tag{3.1}$$

Before we describe how the analog component $E_{\text{a}}^{\text{frame}}$ (Sec. 3.3.2), digital component $E_{\text{d}}^{\text{frame}}$ (Sec. 3.3.3), and communication component $E_{\text{c}}^{\text{frame}}$ (Sec. 3.3.4) are modeled separately, we first discuss a prerequisite of energy modeling: delay estimation (Sec. 3.3.1).

3.3.1 Delay Estimation

The energy consumption, both analog and digital, is correlated with the circuit speed. For example, in the analog domain an operational amplifier (OpAmp) with higher response speed requires larger bias current, which increase the energy consumption (assuming the OpAmp is active over a fixed duration, e.g., when used for an analog frame buffer). The latency of digital units is estimated through cycle-level simulation as described in the previous section. The delay of an analog unit, in contrast, depends on many parameters specific to a fully-designed circuit. We find it cumbersome and error-prone to ask users for input: users often find themselves tuning low-level parameters only to end up with a design that misses the target frame rate. Instead, CAMJ's insight is that each analog unit's delay can be automatically inferred from *the prescribed frame rate*.

Specifically, the fundamental observation that CAMJ relies on is that *the CIS pipeline is designed to never stall*. This is because the input data to the pipeline is generated at a constant rate as the pixel array is exposed to light at the constant speed. If the pipeline ever stalls in a later stage, the frame latency would gradually accumulate, leading to excessively long responsive latency or frame drops. Therefore, CIS designers ensure that the hardware pipeline never stalls. In a fully-pipelined hardware, each pipeline stage must have roughly the same delay; this is the basis of our delay estimation.

Example. Fig. 3.4 shows the pipeline timing for the example in Fig. 3.3. The frame time T_{FR} is 1/FPS, where FPS is the target frame rate. In the diagram, the frame time is the delay between when the pixels of the current frame can be read-out to when the computation of the current frame finishes. The "Binned Pixel Readout" and "ADC" are the two analog units, who share the same delay T_A (i.e., balanced pipeline) to be estimated. The "Edge Detection" is the digital unit, which starts once the second line has been written to the line buffer.

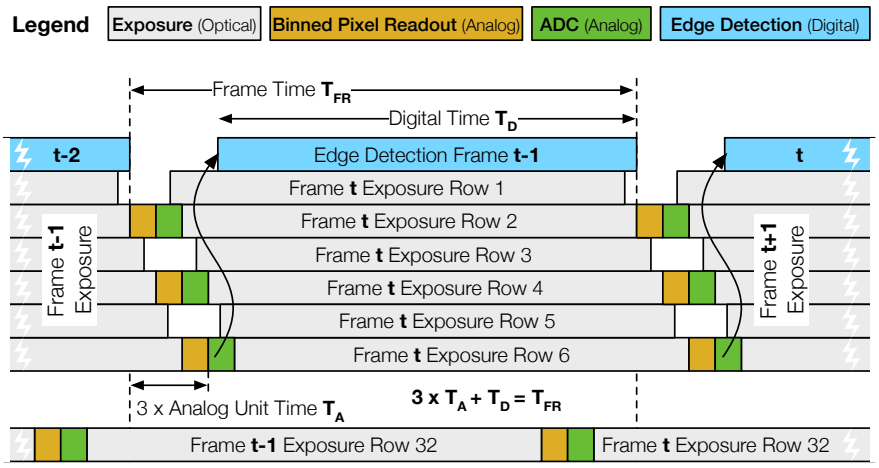


Fig. 3.4: The (not-to-scale) pipeline diagram of the example in Fig. 3.3 when there is no pipeline stall.

To estimate T_A, we first simulate the digital domain to estimate the latency of the entire digital domain (T_D here). Given the frame time T_{FR}, we can then estimate how much time is left for the analog units. In this example, $T_A = \frac{T_{FR}-T_D}{3}$.

CAMJ will analyze the hardware description and, upon detecting potential stalls, asks the user to re-design the hardware to avoid stall. Specifically, CAMJ checks to avoid three scenarios: 1) pixel required is not generated by the producer yet, 2) the memory in-between two stages is full, or 3) the number of access ports in the memory structures is not enough.

3.3.2 Analog Energy Modeling

The analog energy per frame, $E^{\text{frame}}_{\text{analog}}$, is the sum of the energy consumption per access of each A-COMPONENT weighted by the access count to that component. Refer to Tbl. 3.1 for a list of A-COMPONENTs that CAMJ supports.

$$E_{\text{a}}^{\text{frame}} = \sum_{i} \left(E_{\text{a}}^{\text{component}[i]} \times \text{Num}_{\text{access}}^{\text{component}[i]} \right) \tag{3.2}$$

Modeling A-COMPONENTs Access Count. The access count to a A-COMPONENT is the number of times the A-COMPONENT is used per frame. Recall from Sec. 3.2.3 that each A-COMPONENT is part of an Analog Functional Array (AFA). CAMJ leverages the fundamental observation that stencil operations in image processing have regular computation and memory access patterns and, thus, the access count to each A-COMPONENT in the same AFA is the same.

As a result, the access count of a component i is simply the ratio between the total number of operations mapped to the AFA j that contains the component ($\text{Num}_{\text{ops}}^{\text{AFA}[j]}$) and the number of components in that AFA ($\text{Num}_{\text{component}[i]}^{\text{AFA}[j]}$):

$$\text{Num}_{\text{access}}^{\text{component}[i]} = \frac{\text{Num}_{\text{ops}}^{\text{AFA}[j]}}{\text{Num}_{\text{component}[i]}^{\text{AFA}[j]}} \tag{3.3}$$

The numerator is easily derived from the algorithm description of a stencil operation (e.g., calculating the number of MAC operations in a convolution). The denominator is the `num_component` attribute of the AFA (see Fig. 3.3).

Modeling A-COMPONENTs Access Energy. Internally, each A-COMPONENT is built from a set of analog cells, which we call A-CELLs. Modeling per-access energy of an A-COMPONENT requires knowing its cell-level implementation. Expert users can define new cell parameters and/or cell-level implementation of an A-COMPONENT. Absent those, each A-COMPONENT has a default implementation, surveyed from classic and recent CIS designs (Yang et al., 2015; Young et al., 2019; Kaur et al., 2020; Hsu et al., 2020; Park et al., 2021). For instance, a 4T-APS pixel A-COMPONENT consists of a photodiode (PD) A-CELL, a floating diffusion node (FD) A-CELL, and a source follower (SF) A-CELL; a multiplier implemented by switched-capacitor charge re-distribution (Lee and Wong, 2017) consists of a capacitor array A-CELL and an OpAmp A-CELL.

We now describe our energy modeling of A-COMPONENTs, but keep in mind that these design details are abstracted away from typical users. $E_{\text{a}}^{\text{component}}$ is the weighted

sum of the energy consumption of each constituting A-CELL in the component and the access counts to the A-CELL:

$$E_{\text{a}}^{\text{component}[i]} = \sum_{j} E_{\text{a}}^{\text{cell}[j]} \times \text{Num}_{\text{access}}^{\text{cell}[j]} \tag{3.4}$$

Despite large varieties of high-level analog circuits, the A-CELL used for analog in-sensor computing can be categorized to three classes according to circuit characteristics: dynamic A-CELL, static-biased A-CELL, and non-linear A-CELL. They each consume energy in a different way.

❶ Dynamic A-CELL. The energy of a dynamic circuit comes from the charging and discharging of the total capacitance in the circuit:

$$E_{\text{a}}^{\text{cell,d}} = \sum_{i}^{N_c} C[i] \times V_{\text{VS}}[i]^2 \tag{3.5}$$

where N_c represents the total number of capacitance nodes in the dynamic circuit, and $C[i]$ and $V_{\text{VS}}[i]$ are the capacitance and the voltage swing at i^{th} capacitance node, respectively. Typical dynamic A-CELLs include capacitive digial-to-analog converter (CDAC) and passive analog memory.

In Equ. 3.5, V_{VS} is determined by the analog supply V_{DDA} and the number of transistors placed between the analog supply and the ground. The nodal capacitance C is determined by its thermal noise and the computation precision. To guarantee the accuracy of analog computing, the maximum thermal noise should be kept below $\frac{1}{2}$LSB of the data resolution:

$$\sigma_{\text{thermal}} = \sqrt{\frac{kT}{C}}, \quad 3\sigma_{\text{thermal}} < \frac{1}{2}\text{LSB} \tag{3.6}$$

where $\text{LSB} = V_{\text{VS}}/2^{\text{data resolution}}$. Data resolution is algorithm dependent. For example, if $V_{\text{VS}} = 1V$ and the required resolution is 8-bit, the thermal noise should be less than $\frac{1}{3}\frac{1}{2}\frac{1\text{V}}{2^{\text{8-bit}}} = 2.6\text{mV}$, from which C is obtained.

❷ Static-biased A-CELL. The energy of a static-biased circuit comes from the integration of the bias current over a specific time period under the analog supply V_{DDA}:

$$E_{\text{a}}^{\text{cell,s}} = V_{\text{DDA}} \times I_{\text{bias}} \times t_{\text{static}} \tag{3.7}$$

where I_{bias} is the bias current and t_{static} is the time during which the A-CELL is statically biased.

We provide two ways to estimate I_{static} based on circuit details. For A-CELLs where I_{bias} directly drives the load capacitance (e.g. static-biased SF in a pixel), I_{bias} is determined by charging up the load within the given time:

$$I_{\text{bias,1}} = \frac{C_{\text{load}} \times V_{\text{VS}}}{t_{\text{static}}} \tag{3.8}$$

where C_{load} is the load capacitance. The energy is reduced to:

$$E_{\text{a}}^{\text{cell,s}} = C_{\text{load}} \times V_{\text{VS}} \times V_{\text{DDA}} \tag{3.9}$$

For A-CELLs where I_{bias} does not directly drive the load capacitance (e.g. differential operational amplifier in analog memory or discrete-time integrator), I_{bias} can be determined by the classic $\frac{g_{\text{m}}}{I_{\text{d}}}$ method (Jespers, 2010):

$$I_{\text{bias,2}} = \frac{2\pi \times C_{\text{load}} \times \text{GBW}}{g_{\text{m}}/I_{\text{d}}} \tag{3.10}$$

where $\frac{g_m}{I_d}$ is a technology-insensitive factor ranging from 10 to 20 depending on the inversion level of the transistors, and GBW is product between gain (G) and bandwidth (BW).

To use Equ. 3.7 and Equ. 3.10, CAMJ must estimate BW and t_{static}, both of which depend on the A-CELL delay. Specifically, BW is the reciprocal of the A-CELL delay and t_{static} is:

$$t_{\text{static}} = T_A - \sum_{i}^{K} t_i^{\text{cell}} \tag{3.11}$$

where T_A is the delay of the A-COMPONENT containing the A-CELL and is estimated in Sec. 3.3.1; K is the number of cells before the current A-CELL on the A-COMPONENT critical path, and t_i^{cell} is the delay of an A-CELL. Absent timing condition from users, we evenly allocate the A-COMPONENT delay to each A-CELL, based

on the fact that the analog signal uni-directionally flows through the A-COMPONENTs we support so all A-CELLs are on the critical path.

❸ Non-linear A-CELL. For those circuits with non-linear transfer functions, such as ADCs and comparators (which are essentially 1-bit ADCs), they contain both dynamic/static-biased circuit cells and digital logic so it is difficult to estimate the energy from analytical formulas. Instead, we use the ADC's Walden Figure-of-Merit (FoM) plot (Murmann, 2022) surveyed from recently published CIS papers, which shows the ADC's energy-per-conversion vs. its sampling rate. Specifically, given the ADC sampling rate (the reciprocal of the A-CELL delay) we, absent detailed user input, use the median energy-per-conversion at that sampling rate as the estimation. The total energy of non-linear A-CELL is thus obtained by the product of its estimated FoM and the number of required conversions:

$$E_{\text{a}}^{\text{cell,nl}} = FoM \text{ [J/conversion]} \times \text{Num}_{\text{conversion}} \tag{3.12}$$

The access counts to a specific A-CELL are the number of times the A-CELL is used along both the spatial and temporal scale to generate one A-COMPONENT output:

$$\text{Num}_{\text{access}}^{\text{cell}[j]} = \text{Num}_{\text{spatial}}^{\text{cell}[j]} \times \text{Num}_{\text{temporal}}^{\text{cell}[j]} \tag{3.13}$$

For example, if an A-CELL represents an static-biased SF in a pixel, $\text{Num}_{\text{spatial}}^{\text{static-SF}}$ would be the number of SFs in the pixel and $\text{Num}_{\text{temporal}}^{\text{static-SF}}$ would be the number of times the pixel charge is read out (e.g., 2 if correlated double sampling is used to reduce noise (Capoccia et al., 2020)). The access counts information for A-CELLs is hard-coded for each A-COMPONENT and is abstracted away from typical users, but can be updated for a custom design.

3.3.3 Digital Energy Modeling

The digital energy of a frame $E_{\text{d}}^{\text{frame}}$ is the sum of the computation energy of each compute unit $E_{\text{d}}^{\text{comp}}$ and the energy of each memory structure $E_{\text{d}}^{\text{mem}}$:

$$E_{\text{d}}^{\text{frame}} = \sum_{i} E_{\text{d}}^{\text{comp}[i]} + \sum_{j} E_{\text{d}}^{\text{mem}[j]} \tag{3.14}$$

The energy of each compute unit is the product of the energy per cycle $E_{\text{d}}^{\text{cycle}}$ and the number of cycles $\text{Num}_{\text{cycle}}$:

$$E_{\text{d}}^{\text{comp}[i]} = E_{\text{d}}^{\text{cycle}[i]} \times \text{Num}_{\text{cycle}} \tag{3.15}$$

We rely on users to provide $E_{\text{d}}^{\text{cycle}}$, which usually is obtained through HLS/ASIC synthesis flows. The cycle counts, in contrast, are obtained through cycle-level simulation by CAMJ.

The energy consumption for memory accesses is the sum of leakage energy and the dynamic access energy; the latter is the product of the energy consumption for one memory read $E_{\text{d}}^{\text{read}}$ or write ($E_{\text{d}}^{\text{write}}$) and the total number of memory reads (Num_{read}) or writes ($\text{Num}_{\text{write}}$):

$$\begin{aligned} E_{\text{d}}^{\text{mem}[j]} = {} & E_{\text{d}}^{\text{read}[j]} \times \text{Num}_{\text{read}}^{[j]} + E_{\text{d}}^{\text{write}[j]} \times \text{Num}_{\text{write}}^{[j]} \\ & + P_{\text{d}}^{\text{leakage}[j]} \times \frac{1}{\text{FR}} \times \alpha \end{aligned} \tag{3.16}$$

The leakage energy is the product of the leakage power $P_{\text{d}}^{\text{leakage}}$ and the memory active time (i.e., not power-gated), which is a fraction α of the frame time $\frac{1}{\text{FR}}$. Users supply the dynamic read/write energy and leakage power; the access counts and the active time are from the CAMJ simulation.

3.3.4 Communication Energy Modeling

The communication power is dominated by the energy to transfer the data outside the sensor using the energy-hungry MIPI CSI-2 interface and, in the case of 3D-stacking

Table 3.2: Summery of CIS designs for validation, which cover a wide range of design variations. *indicates data not reported in the original papers and are based on our educated guess. Unit of analog memory size is "number of analog values".

CIS	Process Node	Stacked	Analog					Digital	
			Pixel	Memory	PE Operation	PE Position	Op Domain	Memory	PE Size
ISSCC'17 Bong et al. (2017a)	65nm	No	3T APS	20×80	Avg&Add	Column&Chip	Charge&Voltage	160KB	$4 \times 4 \times 64$
JSSC'19 Young et al. (2019)	130nm	No	4T APS	4×240	Logarithmic Sub.	Column	Voltage	-	-
Sensors'20 Choi et al. (2020)	110nm	No	4T APS	No	MAC&MaxPool	Column	Voltage	-	-
ISSCC'21 Eki et al. (2021)	65nm/22nm	Yes	4T APS*	No	-	-	-	8MB	1×2304
JSSC'21-I Hsu et al. (2020)	180nm	No	PWM	No	MAC	Column	Time&Current	-	-
JSSC'21-II Park et al. (2021)	110nm	No	4T APS	No	MAC	Column	Charge	-	-
VLSI'21 Seo et al. (2021)	65nm/28nm	Yes	DPS	No	-	-	-	6MB	-
ISSCC'22 Hsu et al. (2022)	180nm	No	PWM	No	MAC	Column	Time&Current	256B*	1
TCAS-I'22 Xu et al. (2021)	180nm	No	3T APS	No	Mul.&Add	Pixel&Chip	Current	-	-

CIS, the energy of μTSV. In literature the energy of the two interfaces is usually given for energy per Byte. Therefore, the communication energy is given by:

$$E_{\text{c}}^{\text{frame}} = E_{\text{c}}^{\text{mipi}} \times \text{Num}_{\text{Bytes}}^{\text{mipi}} + E_{\text{c}}^{\text{tsv}} \times \text{Num}_{\text{Bytes}}^{\text{tsv}} \tag{3.17}$$

$E_{\text{c}}^{\text{mipi}}$ and $E_{\text{c}}^{\text{tsv}}$ are user supplied with represented data reported in the literature (Liu et al., 2019a). The data volume statistics in both interfaces are generated in CAMJ simulation (based on the algorithm description and algorithm to hardware mapping).

3.4 CAMJ Validation

In this section, we validate CAMJ against real measurement data from nine recent CIS chips (Bong et al., 2017a; Young et al., 2019; Choi et al., 2020; Eki et al., 2021; Hsu et al., 2020; Park et al., 2021; Seo et al., 2021; Hsu et al., 2022; Xu et al., 2021) shown in Table 3.2. These designs span a range of design dimensions including 2D and 3D designs, different process nodes, pixel types, as well as PE designs and memory sizes in the analog and digital domains.

Fig. 3.5 compares the estimated and actual energy per pixel reported in the original papers. Our estimations closely match the measured results, which span several orders

of magnitude, showing both the diversity of the CIS design styles and the wide system power/energy scale that CAMJ can flexibly support and accurately model. Across all designs, CAMJ achieves a Mean Absolute Percentage Error of 7.5% and a Pearson Correlation Coefficient of 0.9999.

Fig. 3.5b – Fig. 3.5j compare the detailed energy breakdown across the nine designs. Whenever possible, we use the circuit parameters reported in the papers. For SRAMs, we use DESTINY (Poremba et al., 2015) to obtain per-access energy. The three papers that perform digital computation all execute DNNs, where a PE is a MAC unit; we use the synthesis result of a 65 nm MAC unit design for per-MAC energy (Bong et al., 2017a), and scale it to other process nodes based on classic CMOS scaling (Stillmaker and Baas, 2017; Sarangi and Baas, 2021).

While overall CAMJ provides an accurate component-level and full-system energy estimation, we find two key reasons behind result mismatches. First, the results are less accurate when CAMJ does not have access to detailed design parameters. For example, the pixel estimation in Fig. 3.5f, Fig. 3.5g, and Fig. 3.5j shows an absolute error of 12.4%, 38.9%, and 33.3%, respectively, due to insufficient circuit parameters on pixel ramp-generator (Fig. 3.5f), pixel parasitic capacitance (Fig. 3.5g), and photodiode voltage swing (Fig. 3.5j). Similarly, the analog PE in Fig. 3.5f and Fig. 3.5b shows an absolute error of 9.3% and 23.7% due to insufficient circuit parameters on sampling capacitance (Fig. 3.5f) and sense amplifier conversion energy (Fig. 3.5b), respectively. In contrast for Fig. 3.5c, where the detailed design parameters are provided for the analog PE, the estimation error is only 0.4%.

The other source of inaccuracy comes from the mismatch between the actual circuit design and CAMJ's default circuit template. For example, the ADCs in Fig. 3.5g and Fig. 3.5h show an absolute difference of 31.7% and 16%, respectively [2]; the original designs use low-power dynamic technique (Fig. 3.5g) whereas CAMJ estimates the energy of ADC based on the FOM survey (Murmann, 2022). The memory in Fig. 3.5j

[2]Both papers consider ADC as a digital unit, which is what we use here.

shows an estimation error of 33.0% because the original design uses customized 8T SRAMs while CAMJ uses standard 6T SRAMs from DESTINY (Poremba et al., 2015), resulting in higher leakage power.

3.5 Related Work

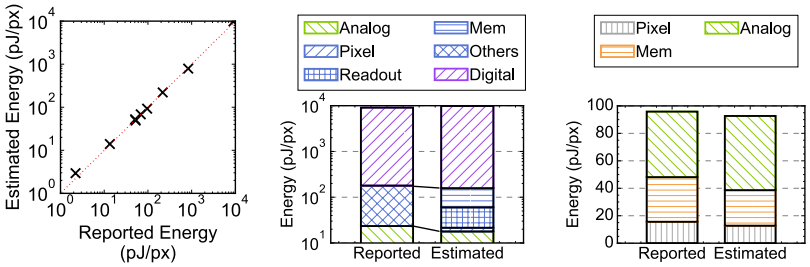
Power Modeling. Power/energy modeling is a cornerstone of architectural exploration. Prior power models of CPUs (Brooks et al., 2000; Li et al., 2009; Shao and Brooks, 2013), GPUs (Hong and Kim, 2010; Leng et al., 2013; Kandiah et al., 2021), and memory (Poremba et al., 2015; Guthaus et al., 2016; Balasubramonian et al., 2017; Pentecost et al., 2022) have enabled a plethora of power/energy optimizations. Fundamentally, CAMJ shares the same, bottom-up modeling methodology, where energy is estimated from access counts and per-access energy. Additionally, CAMJ provides a clean programming interface to integrate other architectural simulators (Shao et al., 2014; Samajdar et al., 2018; Feng et al., 2019; Gao et al., 2017) and memory modeling tools (Poremba et al., 2015; Pentecost et al., 2022) to model bespoke accelerators and memories.

Prior analog power modeling requires either detailed transistor-level parameters (Svensson and Wikner, 2010) or is based on the statistic models of particular analog circuits (Lauwers and Gielen, 2002). Lim et al. (Lim and Horowitz, 2019) decomposes a mixed-signal circuit into basic cells and accelerate the mixed-signal simulation by approximating the transfer function of each cell. CAMJ uses a similar decomposition methodology but specifically targets CIS.

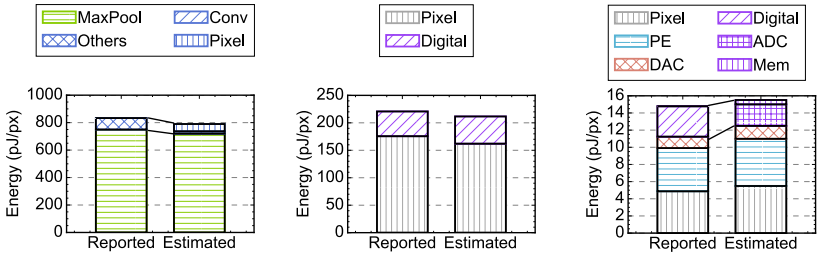
CIS Modeling. No comprehensive CIS modeling framework exists. Two recent papers from Meta use first-order analytical model to estimate the energy of their custom CIS design, i.e., 3D stacking with DPS (Liu et al., 2019a; Gomez et al., 2022). It does not provide the level of flexibility to accommodate general CIS design and architecture exploration as supported by CAMJ.

LiKamWa et al. (2013) provide a coarse-grained CIS power model using the idle and active period/power without considering the hardware implementation details. CAMJ, instead, models the hardware with finer granularity to achieve finer-grained architectural exploration. Kodukula et al. (2021a) cite coarse-grained component energy of typical CIS designs and builds a thermal model. CAMJ, can provide more accurate power/energy modeling that feeds into such thermal model.

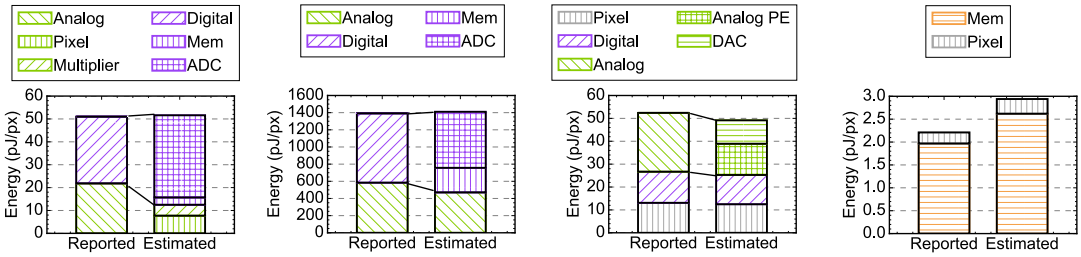
Visual Computing Optimizations. Recent work discusses the possibility of processing inside an CIS to reduce the data transmission cost, e.g., Ed-Gaze (Feng et al., 2022a), Rhythmic Pixel Regions (Kodukula et al., 2021b), Pinkham et al. (2020a), and SplitNets (Dong et al., 2022). All, however, rely on first-order energy models. Many recent visual computing optimizations use motion vectors that can be naturally generated during imaging to simplify downstream vision processing (Zhu et al., 2018b; Feng et al., 2019). It is interesting to explore how motion estimation can be integrated into the CIS using CAMJ. CAMJ can also be integrated with visual computing benchmarks (Huzaifa et al., 2021; Kwon et al., 2022) to study in-CIS computing for different workloads.



(a) Energy Correlation (b) ISSCC'17 (Bong et al., 2017a) (c) JSSC'19 (Young et al., 2019)



(d) Sensors'20 (Choi et al., 2020) (e) ISSCC'21 (Eki et al., 2021) (f) JSSC'21-I (Hsu et al., 2020)



(g) JSSC'21-II (Park et al., 2021) (h) VLSI'21 (Seo et al., 2021) (i) ISSCC'22 (Hsu et al., 2022) (j) TCAS-I'22 (Xu et al., 2021)

Fig. 3.5: Validation results. CAMJ achieves a Pearson correlation coefficient of 0.9999. Several papers lump different components into the coarse-grained "Analog", "Digital", or "Others" categories. We show detailed breakdowns and indicate when the sum of several fine-grained categories in our estimation corresponds to a coarse-grained category in the original papers.

4 EdGaze: In-Sensor Auto ROI for Eye Tracking

While moving computation inside CIS would reduce data communication, processing inside CIS is less efficient compared to off-sensor CMOS processors as illustrated in Sec. 3.1. Therefore, the tradeoff between data communication benefits and compute overheads needs to be addressed. In this chapter, a novel in-sensor algorithm for eye tracking is proposed to show the potential benefits of leveraging in-sensor computing without being compromised by computation overheads.

This chapter begins by proposing EDGAZE which sets an example of in-sensor computing (Sec. 4.1). Sec. 4.2 discusses the mapping between software stages to hardware components in order to achieve optimal energy efficiency and presents a co-designed in-sensor architecture that is tailored for EDGAZE and unleashes the potential of in-sensor computing. Finally, Sec. 4.4 presents a comprehensive evaluation of EDGAZE and its co-designed system.

4.1 EdGaze: In-Sensor Auto ROI for Eye Tracking

Gaze estimation models rely on the geometry of foreground eye parts such as the pupil, iris, and sclera. Full-resolution eye images captured by near-eye cameras usually contain a large chunk of eye muscle/skin that is irrelevant to gaze tracking. We introduce an

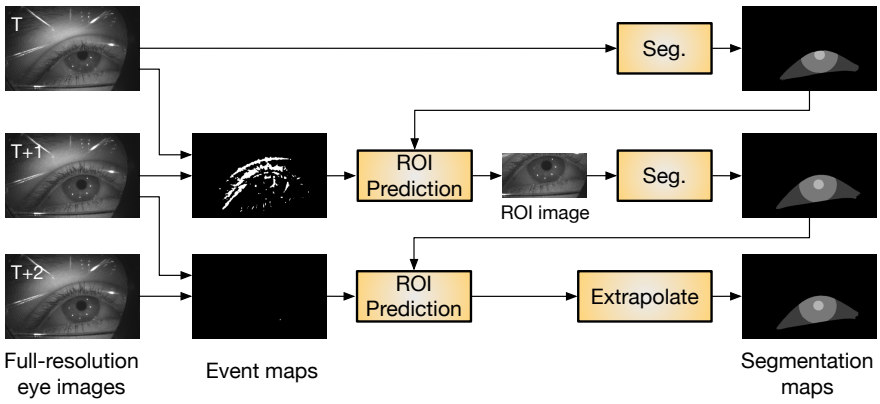


Fig. 4.1: Overview of our event-driven eye segmentation. Time progresses from top to bottom. The segmentation results are used by a common gaze estimation algorithm (Świrski and Dodgson, 2013), which is omitted in the figure. We generate an event map from every two consecutive eye frames; the current event map and previous segmentation map are combined to predict the ROI. If the number of events is small (e.g., shown at time $(T + 2)$), we can simply extrapolate the segmentation result rather than performing a full-blown segmentation.

Auto ROI mode for eye tracking, where we predict and process only the region of interest (ROI) that contains the foreground eye classes needed for gaze estimation. We first provide an overview of the algorithm (Sec. 4.1.1), followed by our feedback-driven, event-based ROI prediction algorithm (Sec. 4.1.2).

4.1.1 Overview

Fig. 4.1 shows a high-level flow diagram of our gaze tracking algorithm. We use a model-based, two-stage algorithm for gaze tracking. This paper's contributions lie in the first stage, i.e., eye segmentation, while relying on a commonly-used gaze estimation model for the second stage (Świrski and Dodgson, 2013). Gaze estimation is relatively more mature and is much less compute intensive that the first stage (regression vs. DNN). Fig. 4.1 thus omits the common gaze estimation part.

Initially at time T, the full-resolution eye frame is processed directly by the segmen-

tation network to generate the segmentation map. The next frame at $(T + 1)$, instead of being processed directly, generates an event map, which is of the same dimension as the original full-resolution image. Each event map pixel is a 0/1 bitmask, indicating whether the corresponding pixel in the original image has a large change in intensity. This process emulates an actual event camera with simplifications tailored to eye segmentation.

The event map provides useful guidance to predict the ROI in the current frame. We define the ROI as the minimal bounding box that encloses the three foreground eye parts, i.e., the pupil, iris, and sclera. Our approach can also apply to other tracking algorithms that use fewer or more segments (e.g., (Yiu et al., 2019; Kothari et al., 2021)).

Events *alone*, however, are not robust enough. When the background eye muscles move significantly and/or when foreground eye parts move little between frames, activated events do not accurately capture the segment boundaries. To improve the robustness, we propose a feedback mechanism, where two important cues from time T (previous frame) are used to augment the ROI prediction.

For cases where the entire eye moves little between frames, e.g., at time $(T + 2)$ in Fig. 4.1, our ROI prediction algorithm would detect the inactivities and opt to extrapolate from the previous segmentation map. Having this mode ensures "activity-proportional" tracking, where little tracking work is done when little activity is observed.

In most cases, only the first frame has to be processed in its full resolution. In rare ($\sim$0.02%) cases where the predicted ROI is physically infeasible (e.g., the top-right corner is to the left of the bottom-left corner), we fall back to the full-resolution mode.

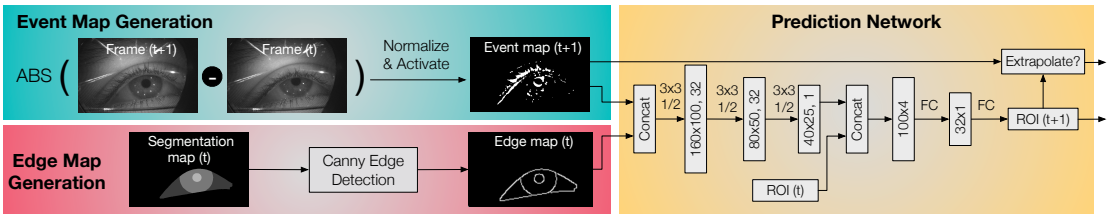


Fig. 4.2: The process of predicting the ROI at time $(T+1)$ consists of three steps. First, we compute the absolute difference of frames at T and $(T+1)$ to generate an event map. Second, we use Canny edge detection on the segmentation map at time T to extract an edge map. Finally, we concatenate the edge map and the event map to form the input to the ROI prediction network. The first three layers are Conv layers with 3×3 kernels followed by a Maxpool layer to reduce each dimension to $1/2$. The output of the last Conv layer is vectorized and concatenated with the ROI from time T (a 1×4 vector). The concatenated vector then goes through two FC layers to generate the predicted ROI of time $(T+1)$. While our algorithm relies on events, the two additional cues of ROI and edge map from the previous frame are critical to the accuracy.

4.1.2 Auto ROI with Event-Driven ROI Prediction

The ROI prediction algorithm has two roles: 1) predicting the ROI of the current eye frame, and 2) deciding whether the current eye frame requires going through a full-fledged segmentation algorithm or can be extrapolated from the previous segmentation map. Fig. 4.2 shows the structure of our ROI prediction network.

The Prediction Algorithm

Intuition The goal of ROI prediction is to filter out background pixels (eye muscles, skin, and eyelids) while leaving only the foreground eye parts, i.e., the pupil, iris, and sclera. We use events to guide the ROI prediction. The intuition is that the background parts do not move as significantly as the foreground eye parts. Thus, activated pixels in the event map mostly correspond to the foreground eye parts and provide useful guidance to the ROI prediction.

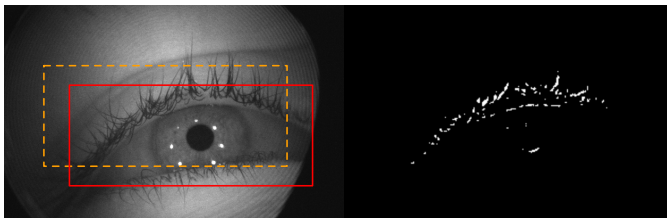


Fig. 4.3: Example of ROI misprediction using only the event map. In the left image, the solid ROI is ground truth, and the dashed ROI is the predicted ROI from using only the event map.

We generate the event map by emulating the high-level process of an event camera (Gallego et al., 2019). Unlike conventional image sensors, which output images at fixed time intervals, event sensors asynchronously respond to local light intensity changes. Each pixel outputs an event when its light intensity change surpasses a predefined threshold. We first calculate the pixel-wise absolute difference, ΔF_{t+1}, between two consecutive frames, F_t and F_{t+1}. Next, each difference value, $\Delta F_t(x, y)$, is normalized by the corresponding pixel value in F_t. The normalized pixel values go through an activation function, Φ, which generates an event when the normalized difference is greater than a predefined threshold value. Mathematically, generating an event map can be expressed as:

$$E_{t+1}(x, y) = \Phi(|F_t(x, y) - F_{t+1}(x, y)|/F_t(x, y), \sigma) \tag{4.1}$$

where $E_{t+1}(x, y)$ is the value at coordinate (x, y) of the event map at time $(T + 1)$, and Φ is the activation function which outputs 1 if the difference is greater than the threshold σ (and 0 otherwise). The threshold is a parameter that can be tuned for a specific application or scenario. Normalizing pixel differences by the previous values mimics the log-scale absolute difference operation done by an actual event camera (i.e., $log(a) - log(b) = log(a/b)$), where the pixel values are naturally in the log scale (Gallego et al., 2019).

Two Feedback Cues While events are largely effective, there are cases where using events alone fails. In particular, when eye muscles/eyelids move significantly and/or foreground eye parts move little, events do not accurately capture the eye boundary anymore, challenging the assumption of using events to predict ROIs.

Fig. 4.3 shows one such example, where the left panel shows the eye frame and the right panel shows the event map (generated from the current and previous frame). The dashed-line box is the predicted ROI using only the event map, whereas the solid-line box is the ground truth ROI. In this example the upper eyelid (irrelevant to gaze estimation) moves upward with the iris (used in gaze estimation); as a result, activated events capture both the eyelids and the iris, leading to a predicted ROI significantly above the actual eye.

To cope with the issue where events in the current event map do not accurately represent the foreground eye parts, we feed the previous ROI back to the prediction algorithm. The intuition is that the ROI of the previous frame provides useful information to predict the ROI in the current frame due to the temporal correlation. Thus, we feed the previous ROI into the ROI prediction algorithm (a DNN), which learns to correlate the previous ROI with the current ROI.

Using the previous predicted ROI, however, has an inherent downside: the ROI prediction errors will accumulate and the predicted ROI will drift over time. To mitigate error accumulation, we use an additional cue that does *not* drift over time — the previous segmentation map. The segmentation map is predicted directly from an actual eye image from the camera; thus, the segmentation map does not drift over time. To reduce data size, we extract an edge map from the segmentation map. The edge is defined as the boundary between two different classes in the segmentation map. We use Canny edge detection (Canny, 1986) to obtain an edge map from a segmentation map. Each pixel value in the edge map is a bitmask that indicates whether the corresponding pixel in the eye frame is a boundary.

Prediction Network With the guidance of the event map and the two feedback

cues, ROI prediction can be very lightweight. Fig. 4.2 shows the network architecture, which contains three convolution (`Conv`) layers and two full-connected (`FC`) layers. The event map and edge map are concatenated first and used as the input to the `Conv` layers due to the 2D nature of the two maps. In contrast, the ROI is a 1×4 vector; thus, it is concatenated with the flattened output of the `Conv` layers before entering the `FC` layers.

To reduce the overhead of the ROI prediction, we downsample the dimensions by 2 for both the event map and the edge map. We normalize the ROI to the image width/height so that each of the four coordinates in the ROI is within the [0, 1] range. We find that it is easier for the network to learn normalized values, as opposed to the absolute coordinates, which vary by camera and scenarios.

Activity-Proportional Segmentation

In cases where the entire eyes do not move across frames, detecting the inactivities and skipping segmentation all together improves the execution speed. This allows our eye tracker to be activity-proportional: no work when no activity is detected. Two issues remain: how to *detect* inactivities and how to *compute* an accurate segmentation map extremely fast for inactive eye frames?

Building on top of our ROI prediction algorithm, we detect inactivity by calculating the event density of the event map inside the predicted ROI. If the event density is lower than a threshold γ, the current frame is deemed inactive, in which case the ROI prediction algorithm sets the extrapolation bit, indicating that no segmentation is to be executed for the current frame. While other inactivity detection schemes are possible, we favor the simple thresholding scheme because the threshold γ provides a useful knob to control the speed-vs-accuracy trade-off, allowing our system to potentially adapt to different application requirements and hardware capability.

We experiment with a range of extrapolation schemes, ranging from simply scaling

the segments from the previous frame to using a neural network to predict how to morph the segmentation from the previous frame. We eventually settled for the simplest scheme, where the previous segmentation result is reused. In retrospect, this scheme is the most robust because it relies the least on the inactivity detection scheme, which is simple thresholding.

4.2 Co-Designed In-Sensor System

So far we have focused on improving the compute efficiency of the eye tracking algorithm. However, the data transmission overhead, both between the image sensor and the processor and between the processor and memory, is non-trivial. A recent study shows that the average energy of data transmission per byte is about 800 times higher than the computation energy per byte (Liu et al., 2019a). Similar observations are made in other recent studies (Kodukula et al., 2021b; Han et al., 2016a). To achieve better energy efficiency, this section first proposes a mapping between algorithm stages to hardware to reduce the overall sensor data transmission (Sec. 4.2.1). Then, a co-designed in-sensor architecture is described that carefully leverages analog computing to further improve the overall system efficiency (Sec. 4.2.2).

4.2.1 Reducing Sensor Data Transmission

The Auto ROI capability of our eye tracking algorithm provides an opportunity to reduce the data transmission overhead. Our idea is to execute part of the algorithm inside the image sensor to reduce the data transmission *volume* and thereby the transmission energy. In-sensor computing has become possible as image sensor vendors have started integrating processing capabilities with the conventional pixel array in one sensor, sometimes in one single chip (Chai, 2020; Kwon et al., 2020; Haruta et al., 2017). A recent example is Sony's IMX 500 image sensor that provides dedicated memories

and a digital signal processor for executing DNNs (Sony, 2020).

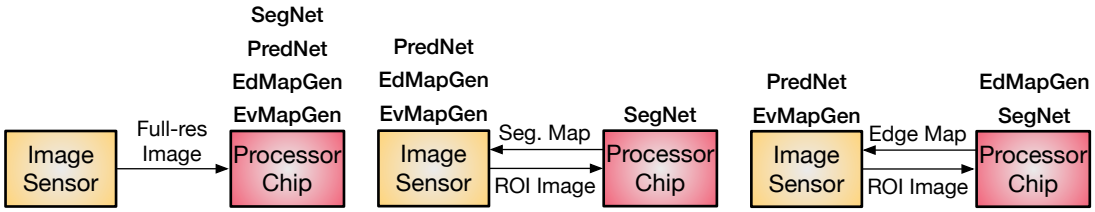
Table 4.1: Total number of Floating Point Operations (FLOPs) and the output data size of the four major components in our algorithm for a 640×400 pixel grayscale input. For this estimate we assume the ROI image is one-third the size of the full resolution, and that half of the frames are extrapolated. We show the full-resolution image size for reference. Recall that data transfer consumes 800 times more energy than computation per byte (Liu et al., 2019a).

Stages	**Event Map Generation**	**Edge Map Generation**	**Prediction Network**	**Seg. Network**	**Full-res Image**
FLOPs (Million)	0.3	1.9	55.4	2641.6	NA
Output Data (KB)	7.8	7.8	41.7	62.5	250

While conceptually simple, reaping the reward of in-sensor computing is not trivial. Computation inside image sensors consumes more energy than that on a backend processor. This is because the semiconductor fabrication technology, quantified by the *process node*, of the sensor usually lags at least one generation behind that of the processor. Today, many commercial processors are fabricated using a 7 nm process node or smaller, but even high-end image sensors still use a 14 nm or 28 nm process node (Yu et al., 2019; Kwon et al., 2020). The computation power consumption increases *quadratically* with respect to the process node (Dennard et al., 1974). Therefore, one must carefully weigh the reduction of data transmission energy against the overhead of computing inside an image sensor on an older process node.

Our goal is thus to map the different components of our eye tracking algorithm to the sensor and processor in such a way that the total energy consumption is minimized. Intuitively, the ideal mapping is one where a trivial amount of computation in the sensor can drastically reduce the data communication.

To identify the optimal mapping, we first quantify the computation cost and the out-



(a) No compute in sensor. (b) Predict ROI in sensor. (c) Optimal mapping.

Fig. 4.4: Different hardware mapping schemes for the four algorithmic components: event map generation (EvMapGen), edge map generation (EdMapGen), ROI prediction network (PredNet), and eye segmentation network (SegNet). The optimal mapping (c) minimizes the total data transmission and computation energy.

put data volume of the four main components in our algorithm: event map generation, edge map generation, prediction network, and segmentation network; the first three comprise the ROI prediction algorithm (Fig. 4.2). Tbl. 4.1 shows the results. Note that each image pixel uses 1 byte; each pixel in the event map and the edge map uses 1 bit; and each segmentation map pixel uses 2 bits (4 classes).

Compared to the segmentation network, the entirety of the ROI prediction algorithm (columns 2–4 in Tbl. 4.1 combined) is much lighter weight, requiring less than 2% of the Floating Point Operations (FLOPs). One straightforward solution would thus be to map the entire ROI prediction algorithm to the sensor while leaving only the segmentation network to execute on the processor chip. This mapping is shown in Fig. 4.4b.

Compared to the mapping in today's systems, where all the computation takes place in the processor (illustrated in Fig. 4.4a), predicting ROI in the sensor reduces the data transmission volume from the full-resolution image to only the ROI image (from sensor to processor) and the segmentation map (from the processor back to the sensor). The reduction in data transmission will outweigh the slight computation energy increase of moving the ROI prediction to the sensor, given the small computation requirements of the ROI prediction algorithm.

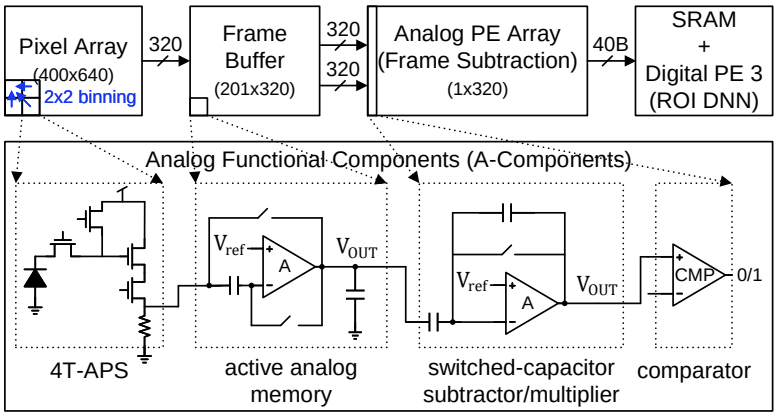


Fig. 4.5: Mixed-signal CIS design for Ed-Gaze. EvMapGen in Fig. 4.4c is moved to the analog domain and PredNet is still mapped to the digital domain.

Interestingly, mapping the ROI prediction network to the sensor is *not* the most energy-efficient mapping. We propose a mapping, shown in Fig. 4.4c, where only the event map generation and the prediction network execute in the sensor while the edge map generation and the segmentation execute on the processor. Compared to Fig. 4.4b, our mapping reduces both the data transmission volume (since the edge map, rather than the segmentation map, is transmitted) and the computation energy (since the edge map is generated in the processor).

4.2.2 Co-Designed In-Sensor Architecture

According to the optimal hardware mapping scheme in Sec. 4.2.1, we explore the benefits of mixed-signal computing and co-design an in-sensor architecture. In our proposed architecture, two stages, EvMapGen and PredNet, in Fig. 4.4 are computed in the analog and digital domains, respectively to achieve best system efficiency.

Fig. 4.5 shows a mixed-signal CIS design for EDGAZE. Inside the pixel array, the 2×2 downsampling is done through pixel binning. The analog frame buffer stores the downsampled analog pixel values, which are read by an analog PE array for frame subtraction in a column-parallel manner. Each analog PE consists of a switched-capacitor

subtractor/ multiplier for absolute subtraction and a comparator for frame delta digitization. Effectively, the output of the comparator is events in EvMapGen stage. The output of the Analog PE array enters the SRAM array, a dedicated PE array then reads the event map from the SRAM array and computes the ROI prediction result in PredNet stage. For a fair comparison and to ensure area overhead is well accounted for, we conservatively set all the capacitors to 100fF. Despite the oversizing, the analog design still yields at least 27% less area than the digital counterpart. In Sec. 4.4.2, we quantitatively evaluate the energy consumption of our proposed CIS design with its pure digital-signal CIS counterpart.

4.3 Evaluation Methodology

4.3.1 Dataset and Ground Truth

OpenEDS 2020 We use the sequential data from the OpenEDS 2020 (Palmero et al., 2020) dataset (Track-2)[1], which contains 200 sequences of continuous frames (29,476 frames in total) gathered from 152 participants with various ethnicities and iris colors. All sequences are captured by an infrared camera at 100 Hz with a 640×400 resolution. The average duration of a sequence is 30 seconds with diverse eye movements such as blinks and saccades (rapid eye movements). The average number of blinks per video is 4.5 (up to 18) and the average number of saccades per video is 7.8.

However, the sequential data in OpenEDS 2020 do not include the segmentation ground truth. To generate the ground truth for our evaluation, we train RITnet (Chaudhary et al., 2019), a state-of-the-art eye segmentation network, on the non-sequential OpenEDS 2019 (Garbin et al., 2019) dataset (100 Hz with a 400×640-pixel resolution), which does have ground truth segmentation labels. We then use the trained RITnet to generate the eye segmentation results for the sequential OpenEDS 2020 data.

[1] We do not use Track-1 data as the videos are only 1–2 seconds long.

The generated eye segmentation results are not perfect; we perform a series of data refinement steps to generate the final ground truth. We first apply the DBSCAN spatial clustering algorithm (Ester et al., 1996) on each initial segmentation result and identify the largest contiguous region as the correct eye region. We then fill the missing holes inside the eye region as a prior study does (Kim et al., 2019). Finally, we manually inspected all sequences of the refined data and removed 15 sequences (out of 200), where the eye segmentations are visibly incorrect. In the end, our sequential dataset contains 185 sequences and 27,431 total frames.

With the eye segmentation ground truth, we then apply a gaze estimation algorithm (Świrski and Dodgson, 2013) commonly used in prior work (e.g., DeepVOG (Yiu et al., 2019)) to generate the gaze ground truth. Our algorithm does not depend on a particular gaze estimation method and, thus, can be integrated with other gaze estimation models (Dierkes et al., 2019). We also use the eye segmentation ground truth to generate the ROI ground truth, which is the bounding box of the foreground segment (pupil, iris, and sclera). We manually verify that using so-defined ground-truth ROIs gives the same gaze results as those from full-frame inference.

4.3.2 Training

We first train a standalone eye segmentation network and a standalone ROI prediction network, and then refine eye segmentation using the ROI prediction results.

We follow prior studies and split the entire dataset into the training/validation/test sets with a 80/20 training/validation split, identical to EllSeg (Kothari et al., 2021). The eye segmentation network is trained using the Adam optimizer, a learning rate of 0.001, and a batch size of 4 for 250 epochs. We use the a loss function that combines both the standard cross-entropy loss and losses that are specialized to the eye structure (Chaudhary et al., 2019). To train the ROI prediction network, we use the Adam optimizer with a learning rate of 0.001, a momentum of 0.9, and the mean squared error loss function.

We train the ROI prediction network for 100 epochs with a batch size of 8.

We then fine-tune the eye segmentation network using ROI images. In this step, the ROI prediction network first predicts an ROI given an eye image; we then crop the input image based on the predicted ROI and fine-tune the segmentation network with the cropped image. This training procedure is similar to networks based on region proposals such as Faster R-CNN (Ren et al., 2015). We use a learning rate of 0.0001, a momentum of 0.9, and 100 epochs.

4.3.3 Baseline and Evaluation Metrics

We primarily compare against eye segmentation methods that are based on DNNs, which are shown to have superior accuracy compared to non-DNN-based segmentation methods (Fuhl et al., 2016a; Akinlar et al., 2021). The baselines are trained using the same procedure as our algorithm.

- `RITnet` (Chaudhary et al., 2019): an encoder-decoder network using DenseNet (Huang et al., 2017) as the backbone. It won first place in the 2019 OpenEDS segmentation challenge (Meta, 2019). As discussed in Sec. 4.3.1, we use RITnet to generate the ground truth for the sequential OpenEDS data. As a result, it is expected that our algorithm will have lower accuracy than `RITnet`.
- `DenseElNet` (Kothari et al., 2021): an ellipse segmentation framework for gaze tracking. `DenseElNet` is originally for ellipse segmentation (three segments). We re-purpose `DenseElNet` to predict four segments by modifying the channel size of the last layer.
- `DeepVOG` (Yiu et al., 2019): a popular encoder-decoder network for eye segmentation. `DeepVOG` was originally designed to generate only two segments (pupil and background). We modified the channel size of the last layer so that it predicts four segments.

We also compare the speed of our algorithms with the baselines. We measure the speed on a state-of-the-art mobile computing platform, Nvidia's Jetson Xavier board (Nvidia, 2019). We use the mobile Volta GPU on the Xavier board for all the networks. The GPU has 512 CUDA cores with a maximum frequency of 1,377 MHz.

4.3.4 Variants

We use two DNN variants of our eye segmentation network, `Ours(S)` and `Ours(L)`. Both are designed with the same architecture, but differ in the channel width and thus provide a speed-vs-accuracy trade-off. In particular, `Ours(S)` has, on average, half the channels of `Ours(L)`. Tbl. 4.2 compares the amount of Floating Point Operations (FLOPs) and parameters of the two networks against the baseline segmentation networks.

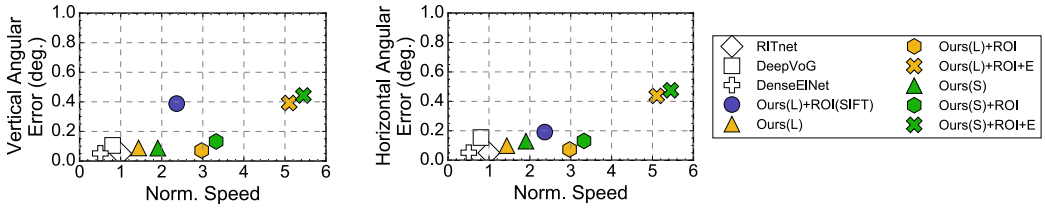
Table 4.2: FLOPs and number of parameters in different eye segmentation networks for an input size of 640×400 8-bit pixels.

Network	**DenseElNet**	**DeepVOG**	**RITnet**	**Ours (L)**	**Ours (S)**
FLOPs (Billion)	53.1	36.5	23.1	2.6	1.2
Norm. FLOPs	45.2	31.1	19.7	2.3	1.0
# of Parameters (Thousand)	3047.3	2835.7	391.0	73.0	30.6
Norm. # of Parameters	99.6	92.6	12.8	2.4	1.0

The total computation requirement of `Ours(L)` is only about 1/9 of `RITnet`, 1/14 of `DeepVOG`, and 1/20 of `DenseElNet`. `Ours(S)` is even lighter weight, with about 45$\times$ fewer FLOPs and 100$\times$ fewer parameters compared to `DenseElNet`. Overall, `Ours(S)` uses only about 30K parameters ($\sim$120 KB).

To tease apart the contributions of the different components of our algorithm, we evaluate two variants on top of eye segmentation:

- `+ROI`: this variant uses only the ROI prediction (Sec. 4.1.2)



(a) Vertical gaze error vs. speedup.

(b) Horizontal gaze error vs. speedup.

Fig. 4.6: The accuracy and speed comparison of different methods. All the subfigures share the same legend. The speedup values are normalized to the speed of `RITnet`. `Ours(S)` and `Ours(L)` are two eye segmentation networks. `+ROI` denotes ROI prediction is enabled. `+E` denotes the extrapolation is enabled. `+ROI(SIFT)` denotes using the SIFT-based ROI prediction.

- `+ROI+E`: this variant uses both ROI prediction (Sec. 4.1.2) and extrapolation (Sec. 4.1.2)

4.4 Evaluation

This section first evaluates the performance of EDGAZE in Sec. 4.4.1. Next, Sec. 4.4.2 evaluates the performance efficiency of the proposed in-sensor architecture.

4.4.1 Evaluation on EDGAZE

Gaze Estimation Our algorithm achieves a 5.5× speedup over the baselines with a sub-0.5° gaze error. Fig. 4.6a and Fig. 4.6b compare the vertical and horizontal gaze errors and the speed of our algorithms with different baselines. The speedups are normalized to the speed of `RITnet`, which runs at 5.4 Hz on a mobile Volta GPU. Note that a 1° error is generally acceptable for gaze tracking (Kar and Corcoran, 2017).

`RITnet`, `DenseElNet`, `Ours(L)`, and `Ours(S)` keep the absolute error rate

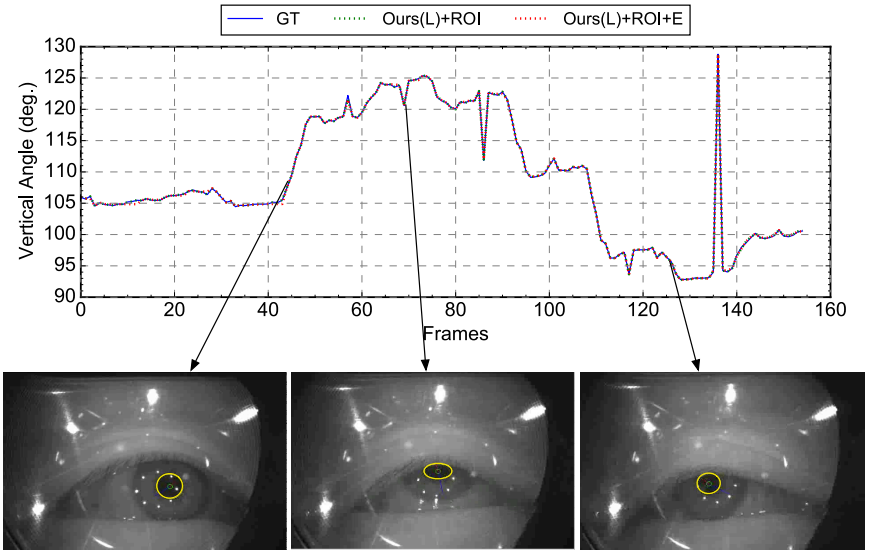


Fig. 4.7: Gaze estimation results over one sequence of frames. `Ours(L)` robustly tracks the ground truth. The bottom panel shows three representative cases: eye moves right, just before a blink, and eye moves up-left, respectively.

below $0.1°$ in both the vertical and horizontal direction. They are all more accurate than `DeepVOG`. In particular, `Ours(S)` only has 0.04 and 0.05 higher gaze error than `RITnet` in each direction, but is $1.9\times$ faster. Note that this little gaze error loss is achieved with a network that is $12.8\times$ smaller (Tbl. 4.2).

By operating on ROI images when possible, `Ours(L)+ROI` improves the speedup over `RITnet` to $3.0\times$. Interestingly, `Ours(L)+ROI` achieves better accuracy than `Ours(L)`. Further inspection of the data shows that this is because by using only the ROI image for eye segmentation, we remove potential noise in the non-ROI region in the input eye image. Using activity-proportional segmentation, `Ours(L)+ROI+E` and `Ours(S)+ROI+E` further improve the speedup to $4.2\times$ and $5.5\times$, respectively, while both retaining an angular error rate within $0.5°$. The event density threshold used for extrapolation is set to 0.1%.

To further confirm the robustness of our system, Fig. 4.7 compares the frame-by-frame gaze results of `Ours(L)`, `Ours(L)+ROI+E`, and the ground truth. `Ours(L)`

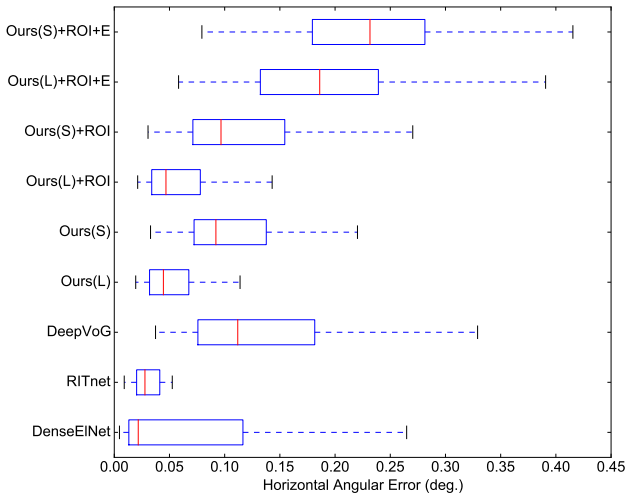


Fig. 4.8: Distributions of horizontal gaze error across as boxplots, which plot the median, 25th-percentile, 75th-percentile, the min, and max of the angular errors. `RITNet` is the most accurate because it is used to obtain the ground truth (see Sec. 4.3.1). Even the most inaccurate variant of our system, `Ours(S)+ROI+E` has a worst-case accuracy below $0.5°$, which is generally regarded as an acceptable error bound for eye tracking (Kar and Corcoran, 2017).

virtually matches the ground truth, and `Ours(L)+ROI+E` has slight deviations (e.g., around frame 10). We show three representative gazes in the bottom panel, where the eye moves right, blinks, and moves up left.

Finally, for a comprehensive analysis, we show the gaze error distributions across all the evaluated frames in Fig. 4.8. Not surprisingly, `RITNet` has the most compact distribution, because it is used to obtain the ground truth (see Sec. 4.3.1). `Ours(L)` has significantly better accuracy distribution compared to `DeepVOG`. Compared to `DenseElNet`, `Ours(L)` has $0.02°$ lower average accuracy, but also has much better worst-case accuracy, as indicated by the significantly shorter tail. As we use a smaller network and exploit more speed-enhancing techniques, the error distribution generally shifts toward the right, but even the most inaccurate variant, `Ours(S)+ROI+E`, has a worst-case accuracy lower than $0.5°$, which is regarded as an acceptable error bound

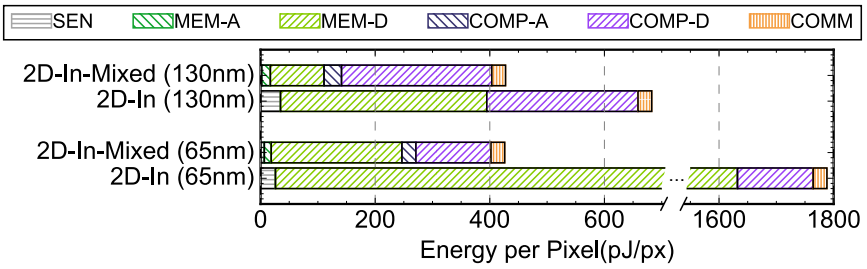


Fig. 4.9: Energy comparison between mixed-signal in-sensor computation and fully-digital in-sensor computation on Ed-Gaze. COMP/MEM-D: digital compute and memory; COMP/MEM-A: analog compute and memory.

for eye tracking (Kar and Corcoran, 2017).

4.4.2 Evaluation on In-Sensor Architecture

Computing inside CIS reduces the data transmission cost by consuming pixel data inside the sensor. To explore the benefits, we evaluate the potential benefit of in-sensor computing by moving the ROI generation inside the sensor.

We use CAMJ (Ma et al., 2023) to evaluate two hardware configurations:

- **2D-In (H)**: a 2D CIS fabricated in the **H** process node; the entire execution, EvMapGen and PredNet, is performed inside the CIS in the digital domain (Fig. 4.4c).
- **2D-In-Mixed (H)**: a 2D CIS in the **H** process node, where EvMapGen is implemented in the analog domain while PredNet is implemented in the digital domain (Fig. 4.4c).

Fig. 4.9 compares **2D-In-Mixed** and **2D-In**. Moving the first stages of the Ed-Gaze algorithm to the analog domain reduces the energy by 38.8% and 77.1%. The energy reduction comes from two sources: removing the ADCs (indicated by lower SEN) and replacing SRAMs in the first two stages with analog buffers (indicated by

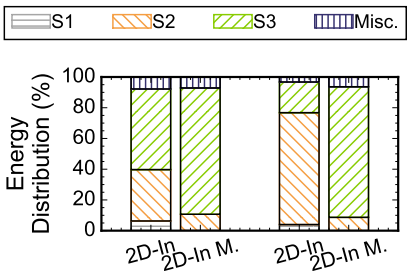


Fig. 4.10: Normalized energy breakdown among the three stages (S1, S2, S3).

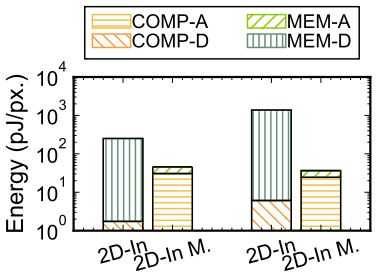


Fig. 4.11: Energy breakdown of first two stages.

lower MEM-D). The reduction in MEM-D is particularly significant for the 65nm node, where the SRAM leakage power is high. To corroborate the results, Fig. 4.10 shows the normalized energy breakdown among the three stages (S1, S2, and S3). S3 (DNN) becomes the dominant stage after moving first two stages into analog domain, showing the effectiveness of analog processing.

Interestingly, the energy reduction is obtained when the compute energy of the first two stages slightly increase. Fig. 4.11 shows the energy breakdown of the first two stages. While the memory energy reduces, the compute energy increases in the mixed-signal mode. This is because to maintain an 8-bit precision the OpAmp consumes too much energy (Ma et al., 2023). A caveat is that the analog design presented here, which uses active switched-capacitor circuits, is based on our specific implementation choice. It is conceivable that different designs would yield different efficiency results.

4.5 Related Work

Feature Extraction While historically hand-crafted, geometric features are popular (Hansen and Pece, 2005; Świrski et al., 2012; Fuhl et al., 2016b), DNN algorithms have recently been shown to be more robust and accurate (Fuhl et al., 2016a; Chaudhary et al., 2019; Yiu et al., 2019; Kim et al., 2019; Kothari et al., 2021). In particular, extracting features through eye segmentation (Chaudhary et al., 2019; Yiu et al., 2019;

Kim et al., 2019; Kothari et al., 2021) is by far the most widely used method.

An eye segmentation algorithm usually classifies the pixels in an eye image into different classes. The resulting segmentation map has the same dimensions as the eye image, and each pixel value in the map represents the class ID of that pixel. Our algorithm uses a similar approach as DeepVOG (Yiu et al., 2019), RITnet (Chaudhary et al., 2019), OpenEDS (Meta, 2019), and Kim et al. (Kim et al., 2019), known as "part segmentation," which segments the eye image into four parts: pupil, iris, sclera, and background. In contrast, Ellseg (Kothari et al., 2021) and Wang et al. (Wang et al., 2021) use elliptical segmentation, which predicts the ellipses of the pupil and iris.

Prior studies focus primarily on accuracy. In contrast, we focus on *compute efficiency*, and show that tracking can be five times faster with little sacrifice in accuracy. We note that while we demonstrate our ROI prediction on part segmentation-based eye tracking, the idea applies generally to tracking algorithms using other features.

Event Camera Event cameras operate each pixel independently and asynchronously (Gallego et al., 2019). Each pixel gets activated when its intensity change surpasses a predefined threshold. The response is called an "event", which includes the pixel coordinates, a timestamp, and a polarity value. Because of the increased hardware complexity, the resolution of event cameras is typically lower than that of conventional cameras (Gallego et al., 2019), but event cameras have a much higher frame rate (upward of tens of thousands of Hz) since they produce only (sparse) events occasionally rather than (dense) pixels regularly.

Event cameras appear in a range of vision and robotics tasks such as object tracking (Ramesh et al., 2018), localization (Weikersdorfer et al., 2013), and reconstruction (Kim et al., 2016). Recent work has also started using event cameras for eye tracking (Damian et al., 2007; Angelopoulos et al., 2020), and is able to achieve a 10K Hz frequency. While extremely high tracking frequency is needed when capturing precise eye movement (e.g., foveated rendering), a lower frame rate provided by conventional cameras is sufficient for many eye tracking use cases, e.g., eye communication sys-

tem for disability (Caligari et al., 2013). Instead of using event camera hardware, we emulate event camera output from our conventional sensor using software.

ROI Computation ROI is widely used to reduce the overall computation and data transmission (Girshick et al., 2014; Ren et al., 2015; Girshick, 2015; Kong et al., 2016; He et al., 2017a; Zhu et al., 2018b; Feng et al., 2019; Mudassar et al., 2019; Kodukula et al., 2021b). Classic work such as fast R-CNN (Girshick, 2015) use dedicated region proposal networks that are computationally heavy. Other approaches use simple extrapolation (Zhu et al., 2018b; Feng et al., 2019; Mudassar et al., 2019), which we find insufficient for eye tracking, because the objects (eyes) move rapidly. Many image sensors provide an ROI output mode (OnSemi, 2015; OmniVision, 2021), but rely on users to provide the ROI coordinates. Sony built a sensor that automatically detects an ROI to drive spatial resolution and exposure modulation (Kumagai et al., 2018).

Our contribution is a lean and accurate ROI prediction algorithm tailored to eye tracking. We show that software-emulated events, combined with edge information from previous segmentation results, can effectively predict eye movement and, by extension, the ROI.

5 ASV: Leveraging Temporal Correlations to Avoid Redundant Computation

While in-sensor computing can effectively reduce the energy from data communication in vision system pipelines, computation in processing is inevitable. In particular, today's state-of-the-art algorithms have increasingly high computation costs, and reducing such computation costs in vision processing has become a pressing issue for many mobile vision applications. Failing to process an image frame in time will inevitably result in a frame dropping which leads to an unsatisfied user experience. This chapter proposes to exploit the inter-frame correlations across the temporal domain and effectively reduce the redundant computation. Specifically, this chapter demonstrates leveraging temporal correlation can be applied to depth estimation, achieving significant speedup without any accuracy sacrificing.

In this chapter, necessary backgrounds are first introduced in Sec. 5.1. Then, Sec. 5.2 and Sec. 5.3 present our proposed algorithm that speedups the depth estimation with its companion hardware design in Sec. 5.4. Next, Sec. 5.5 and Sec. 5.6 quantitatively evaluate the proposed system. Lastly, Sec. 5.7 discusses related work.

5.1 Background

We first describe the scope of our work: vision-based systems that extract 3D information from 2D stereo images (Sec. 5.1.1). We then introduce the necessary background of stereo vision algorithms, including both classic hand-crafted algorithms and contemporary stereo DNNs (Sec. 5.1.2).

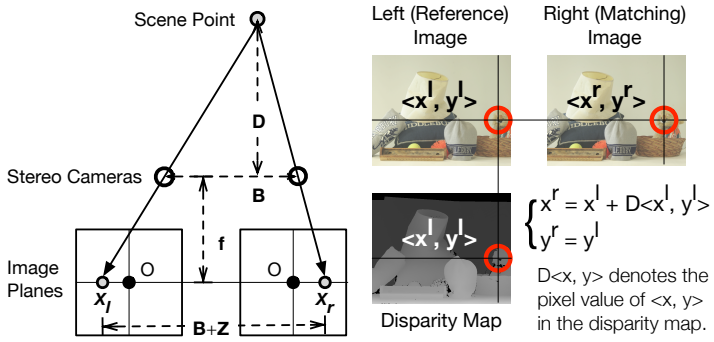
5.1.1 Depth Sensing

There are two essential methods to extract depth information: passive sensing and active sensing. Passive sensing techniques observe the environment, primarily through cameras, and infer depth using computer vision algorithms. In contrast, active sensing techniques transmit signals and analyze the response to calculate depth; examples include structured light (Wang et al., 2012) and LiDAR (Weitkamp, 2006).

This paper focuses on camera-based passive sensing. Compared to alternatives such as LiDAR, cameras are much cheaper and less bulky (Lee, 2017). As a result, camera-based depth sensing is widely adopted in systems such as autonomous vehicles and AR headsets. According to Allied Market Research, the adoption of stereo cameras is expected to grow 60.4% by 2020 (Patil, 2020). The recent industry trend of integrating dedicated stereo vision accelerators into mobile SoCs (e.g., Movidius (Intel, 2017) and Nvidia (Nvidia, 2018)) further underlines the significance of stereo vision for depth sensing.

5.1.2 Depth From Stereo

Triangulation The key idea behind stereo depth estimation is that a single physical scene point projects to a unique pair of pixels, via two observing cameras; the horizontal displacement between the two pixels captured on the left and right image planes is inversely proportional to the distance of the point from the observer (i.e., the



(a) Triangulation. (b) Disparity Map.

Fig. 5.1: "Depth from stereo" illustration: given an image pair, *stereo matching* algorithms first generate the disparity map (b), from which depth is then calculated through *triangulation* (a). Triangulation is computationally trivial; this paper focuses on optimizing stereo matching algorithms.

depth). Fig. 5.1a illustrates this process, where the scene point is captured at position x^l and x^r on the left and right image planes, respectively. Using similar triangles, the depth D is calculated by:

$$D = Bf/Z, \tag{5.1}$$

where f is the focal length of the cameras, B is the distance between the two camera lenses, and Z is the *disparity* $x^r - x^l$, i.e., the horizontal displacement between the two corresponding pixels in the left and right images. This process is widely known as *triangulation* (Hartley and Zisserman, 2003; Szeliski, 2010).

Stereo Matching and Disparity Map Since both B and f are camera intrinsic parameters, the key to triangulation is to calculate the disparity Z. Given the left (reference) image and the right (matching) image, we must find the pixels in each image that are the projections of the same physical point, a process also known as *stereo matching*. In the end, stereo matching generates a "disparity map", whose ¡x, y¿ coordinates are taken to be coincident with the pixel coordinates of the reference image. Fig. 5.1b

shows one such example, in which the correspondence between a pixel $\langle x^l, y^l \rangle$ in the left image and a pixel $\langle x^r, y^r \rangle$ in the right image is given by:

$$x^r = x^l + D^{\langle x^l, y^l \rangle}, \quad y^r = y^l, \tag{5.2}$$

where $D^{\langle x^l, y^l \rangle}$ denotes the pixel value at $\langle x^l, y^l \rangle$ in the disparity map. Note that the compute cost of triangulation is trivial (Equ. 5.1), and thus we focus on stereo matching.

5.2 Invariant-based Stereo Matching

This section introduces our new *invariant-based stereo matching algorithm* (ISM). The key idea of ISM is to exploit the *correspondence invariant* between the stereo images over time. After introducing the high-level concept (Sec. 5.2.1), we then describe the detailed algorithm (Sec. 5.2.2), and discuss important algorithmic design decisions (Sec. 5.2.3).

5.2.1 Overview

Stereo matching produces a disparity map (Fig. 5.1b), from which depth information is easily obtained through triangulation (Fig. 5.1a). Classic stereo matching algorithms generate the disparity map by matching pixels/features in the left (reference) frame with pixels/features in the right (matching) frame, typically by searching in a finite window. However, the accuracy of search-based algorithms is sensitive to the heuristics used in the search, such as feature selection, search window size, matching criterion, etc. In contrast, DNN approaches largely avoid heuristics and instead directly learn the matching pairs. Unfortunately, DNNs come at the cost of a massive increase in compute requirement.

Instead of the binary choice between DNNs and conventional search-based algorithms, we use DNNs to *guide* the search process of classic methods. The key ob-

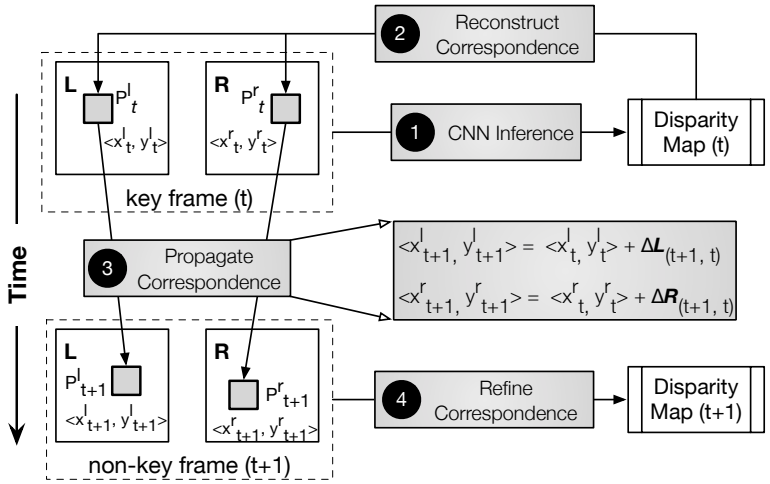


Fig. 5.2: The ISM algorithm obtains correspondences in key frames using DNNs, and propagates the correspondences to non-key frames to guide the cheap correspondence search. Time progresses from top to bottom in the figure.

servation is that two matched pixels, one from the left image and the other from the right image, correspond to the same point in the physical world. While the locations of the two pixels move from frame to frame, they are always projections of the same scene point, and therefore are always a matched pair in any frame. In other words, the geometric correspondence relationship between two matched pixels is invariant.

Our new stereo matching algorithm, ISM, exploits this *correspondence invariant* by operating in two modes. It obtains stereo correspondences on "key frames" through accurate but compute-intensive DNNs. The correspondences are then propagated to subsequent non-key frames as good initial guesses to guide the cheaper search-based methods. By combining learnt correspondences with search-based methods that explicitly model the physical world, ISM reduces the total compute cost while retaining DNN-like accuracy.

5.2.2 Algorithm

We illustrate ISM in Fig. 5.2. ISM consists of four main components. ISM runs DNN inferences (❶) on key frames to obtain pixel correspondences, which are used to guide feature matching on non-key frames (❷, ❸, and ❹).

❶ DNN Inference Assuming the left and right frames at timestep t are regarded as key frames, ISM performs DNN inference to generate a disparity map for the left image, in which each pixel value represents the disparity (i.e., Z in Fig. 5.1a) of each pixel in the left frame. In conventional DNN approaches, this disparity map is used only for triangulation (not shown in the figure) to estimate depth information and is discarded after the depth map is generated.

❷ Reconstruct Correspondences Instead of discarding the disparity map, ISM uses it to identify the correspondences in the left and right frames. As per the definition of disparity (Equ. 5.2), every $\langle x_t, y_t \rangle$ pixel in the disparity map with the value $D_t^{\langle x_t, y_t \rangle}$ indicates that the $\langle x_t, y_t \rangle$ pixel in the left frame (P_t^L) and the $\langle x_t + D_t^{\langle x,y \rangle}, y_t \rangle$ pixel in the right frame (P_t^R) form a correspondence pair. By iterating through all the pixels in the disparity map, ISM identifies all the correspondence pairs in the left and right frames at timestep t.

❸ Propagate Correspondences A new pair of frames arrives at the next timestep $(t+1)$. ISM exploits a well-known observation that pixels in consecutive video frames are highly-correlated in time. For instance, P_t^L has moved to P_{t+1}^L, and P_t^R has moved to P_{t+1}^R. Critically, since P_t^L and P_t^R are a correspondence pair projected from a scene point, P_{t+1}^L and P_{t+1}^R must correspond to the same point, and hence highly likely to also be a correspondence pair at timestep $(t+1)$.

The exact coordinates of P_{t+1}^L and P_{t+1}^R can be obtained through a *motion estimation* (ME) algorithm. For each pixel in the left (right) frame, the ME algorithm generates a motion vector $\Delta P_{(t+1,t)}^L$ ($\Delta P_{(t+1,t)}^R$), representing the displacement between the pixel in

frame t and frame $(t+1)$. Thus:

$$P^L_{t+1} = P^L_t + \Delta P^L_{(t+1,t)}$$

$$P^R_{t+1} = P^R_{t+1} + \Delta P^R_{(t+1,t)}$$

❹ **Refine Correspondences** Given the correspondence pairs (e.g., P^L_{t+1} and P^R_{t+1}) at timestep $(t+1)$, ISM then calculates the disparity map at $(t+1)$. If the motion estimation from t to $(t+1)$ is precise, the propagated correspondence pairs at $(t+1)$ are also precise. Accordingly, the disparity map could be simply obtained by calculating the horizontal offsets between all the correspondence pairs. For instance, given the correspondence pair P^L_{t+1} and P^R_{t+1}, the disparity at $\langle x^l_{t+1}, y^l_{t+1}\rangle$ in the disparity map would be $x^r_{t+1} - x^l_{t+1}$.

In reality, motion estimation is imperfect due to various visual artifacts such as occlusion and fast motion (Liu et al., 1998). Thus, the correspondences propagated from t are a noisy estimate of the true correspondences at $(t+1)$. To further refine the estimate of $(t+1)$ in ISM, we use classic *correspondence search*, and initializes the search window with the propagated correspondences. This allows ISM to avoid compute-intensive DNNs on non-key frames without sacrificing accuracy.

5.2.3 Algorithmic Design Decisions

Computing non-key frames requires reconstructing, propagating, and refining correspondences. Reconstructing correspondences has little overhead. The cost of propagating correspondences is dominated by motion estimation, and the cost of refining correspondences is dominated by the correspondence search. Thus, we must carefully choose the motion estimation and correspondence search algorithms such that the compute cost is much lower than DNNs with little accuracy loss. We discuss algorithmic choices below.

Motion Estimation The literature is rich with motion estimation algorithms, which differ in the coverage and densities of estimated motion. The disparity map

in stereo matching should ideally be calculated on a per-pixel basis across the frame, so as to enable fine-grained depth estimation. This requirement rules out many classic motion estimation algorithms such as block matching (BM) (Jakubowski and Pastuszak, 2013), and sparse optical flow (Lucas and Kanade, 1981; Horn and Schunck, 1981). BM estimates motion at the granularity of a block of pixels, and thus does not provide the pixel-level motion that stereo vision requires. Sparse optical flow algorithms such as Lucas-Kanade (Lucas and Kanade, 1981) and Horn-Schunck (Horn and Schunck, 1981) only provide pixel-level motion for feature points such as corners, and do not cover all the frame pixels.

Instead, we use a *dense optical flow* algorithm, specifically the Farneback algorithm (Farnebäck, 2002, 2003), for motion estimation. Farneback generates per-pixel motion for all the pixels, and is computationally efficient. 99% of the compute in Farneback is due to three operations: Gaussian blur, "Compute Flow", and "Matrix Update". Gaussian blur is inherently a convolution operation that convolves a Guassian kernel (2D matrix) with the image. The latter two are point-wise operations that resemble the activation function in DNNs. Thus, motion estimation in the ISM algorithm can be computed using a DNN accelerator to simplify the hardware design.

Correspondence Search ISM performs correspondence search to refine the initial correspondence estimation propagated through motion. Correspondence search algorithms have been well-studied in the classic computer vision literature (Scharstein et al., 2001; Brown et al., 2003), and generally fall into two categories: local methods and global methods. At the cost of higher compute demand, global methods provide higher accuracy by minimizing the pixel motion inconsistencies across the entire image. However, with the initial correspondences propagated through key-frames, we find that local methods suffice.

In particular, we leverage the block matching algorithm (Jakubowski and Pastuszak, 2013) for local correspondence search. For each pixel in the left image (e.g., P^L_{t+1} in Fig. 5.2), ISM uses the block of pixels surrounding it to search in a 1D window in the

right image in order to find the closest match. The search window is centered around the initial correspondence estimation (e.g., P^R_{t+1} in Fig. 5.2). We use the sum of absolute differences (SAD) cost function. The horizontal offset between the two matched blocks is the disparity for P^L_{t+1}.

Similar to optical flow, the block matching algorithm has a "convolution-like" structure (Qadeer et al., 2013); the block in the left image is equivalent to a kernel, and the search window in the right image is equivalent to the input image. The only difference is that block matching computes the SAD between the input feature map and the kernel ($\sum_{i=1}^{N} |a_i - b_i|$) as opposed to the dot product in canonical convolution ($\sum_{i=1}^{N} a_i b_i$). Thus, the correspondence search can share the same architecture as DNNs and optical flow.

Compute Cost Due to our algorithmic choices, computation on non-key frames is much cheaper than key-frames. For instance, for a typical qHD frame (960×540), computating a non-key frame requires about 87 million operations while stereo DNN inference (key frame) requires about $10^2\times$–$10^4\times$ more arithmetic operations. Thus, ISM leads to significant performance and energy improvements by avoiding DNN inference altogether in non-key frames.

5.3 Deconvolution Optimizations

While the ISM algorithm removes DNN inference in non-key frames, DNNs remain critical for generating initial key frame correspondences. This section describes optimizations for stereo DNNs, in particular the dominant deconvolution layers. We propose novel software-only optimizations that mitigate the compute overheads in deconvolution (Sec. 5.3.1), while capturing unique data reuse opportunities (Sec. 5.3.2).

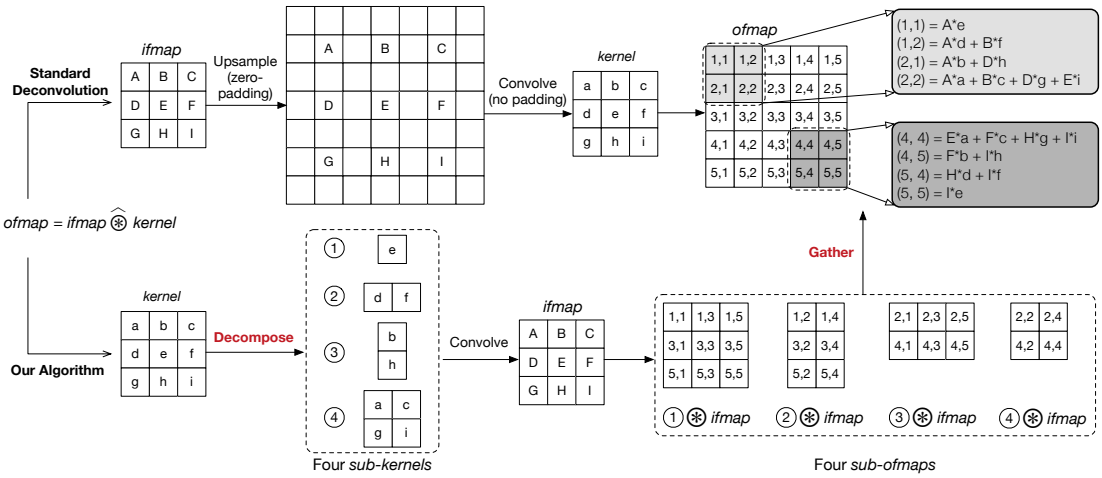


Fig. 5.3: Translating deconvolution into multiple convolutions. Standard deconvolution first upsamples the *ifmap* before convolving with the kernel. Note that this example assumes the upsampled *ifmap* is not further padded before the convolution, i.e., a 7×7 *ifmap* results in a 5×5 *ofmap*. Our translation algorithm holds regardless of padding.

5.3.1 Deconvolution Transformation

Deconvolution layers on average contribute to 38.2% (50% max) of the total MACs in stereo DNNs (Feng et al., 2019). Due to the inherent sparsity of deconvolution, a naive mapping to hardware results in over 75% of redundant computations due to one or more zero operands. Deconvolution is also used in Generative Adversarial Networks (GANs), and recent studies have proposed specialized hardware specifically for deconvolution (Song et al., 2018; Yazdanbakhsh et al., 2018). In contrast to previous studies, we propose a *purely algorithmic transformation* that eliminates inefficiencies due to sparsity. We show that an inherently sparse deconvolution layer can be translated to a series of dense convolutions, which then effectively map on to existing DNN accelerators. We next explain the inefficiencies in deconvolution, and then describe our algorithmic transformations.

Standard Deconvolution The standard process (Fig. 5.3) deconvolves a 3x3 input feature map (*ifmap*) with a 3x3 kernel. The *ifmap* is first upsampled with zero padding,

before being convolved with the 3x3 kernel to generate an output feature map (*ofmap*). Note that the upsampling step essentially performs disparity refinement, which is fundamental to general stereo DNNs, rather than being specific to a particular network. The zeros in the upsampled *ifmap* leads to redundant computation and memory traffic.

A key characteristic of deconvolution is that different elements in the *ofmap* are calculated in different "patterns." Consider the first 2×2 outputs in the *ofmap*: (1, 1), (1, 2), (2, 1), and (2, 2). Each of the four outputs is generated using a different set of elements from the kernel. For instance, (1, 1) requires only e while (1, 2) requires d and f. Critically, there are only four different patterns, which are repeated across the *ofmap*. Pixels (4, 4) and (2, 2) are calculated using the same elements from the *kernel*, as are (1, 1) and (5, 5), (1, 2) and (5, 4), as well as (2, 1) and (4, 5). Due to the various patterns needed to generate different output elements, deconvolution is clearly an "irregular" operation. Prior work (Yazdanbakhsh et al., 2018) exploits the four unique computation patterns by augmenting a conventional DNN accelerator with custom hardware units.

Deconvolution Transformation In contrast, we find that existing DNN accelerators already provide the necessary architectural substrate to efficiently execute the four different patterns. The key is to recognize that the four computation patterns are essentially four different convolutions, each convolving the original *ifmap* with a distinct kernel that is part of the original kernel. For instance, (2, 2), (2, 4), (4, 2), and (4, 4) are generated by convolving $\left[\begin{smallmatrix} a & c \\ g & i \end{smallmatrix}\right]$ with *ifmap*. More generally, the deconvolution in Fig. 5.3 is calculated as:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \widehat{\circledast}\, I = \mathcal{G}\left(\begin{bmatrix} e \end{bmatrix} \circledast I, \begin{bmatrix} d & f \end{bmatrix} \circledast I, \begin{bmatrix} b \\ h \end{bmatrix} \circledast I, \begin{bmatrix} a & c \\ g & i \end{bmatrix} \circledast I\right)$$

where $\widehat{\circledast}$ denotes the deconvolution operation, $\circledast$ denotes the standard convolution operation, I is the *ifmap*, and $\mathcal{G}$ is a gather operation to assemble the *ofmap* from the results of the four convolutions. $\mathcal{G}$ is simply implemented as a set of load operations to

the on-chip buffer. Essentially, our algorithm decomposes the original 3×3 kernel into four sub-kernels, each requiring a smaller dense convolution with the original *ifmap*, which can be executed efficiently on a conventional DNN accelerator.

This transformation generalizes to kernel shapes other than 3×3. Formally, a 2D kernel K with a dimension $K_H \times K_W$ will be decomposed into four sub-kernels (S_0, S_1, S_2, S_3):

$$
\begin{aligned}
S_0^{(i,j)} &= K^{(2i,2j)} && i \in [0, \lceil K_H/2 \rceil), j \in [0, \lceil K_W/2 \rceil) \\
S_1^{(i,j)} &= K^{(2i+1,2j)} && i \in [0, \lfloor K_H/2 \rfloor), j \in [0, \lceil K_W/2 \rceil) \\
S_2^{(i,j)} &= K^{(2i,2j+1)} && i \in [0, \lceil K_H/2 \rceil), j \in [0, \lfloor K_W/2 \rfloor) \\
S_3^{(i,j)} &= K^{(2i+1,2j+1)} && i \in [0, \lfloor K_H/2 \rfloor), j \in [0, \lfloor K_W/2 \rfloor)
\end{aligned}
$$

where $S_*^{(i,j)}$ is the element (i, j) in a particular sub-kernel, and $K^{(*,*)}$ is an element in the original kernel K. For instance, $S_0^{(i,j)} = K^{(2i,2j)}$ means that element (i, j) in the first sub-kernel comes from element $(2i, 2j)$ in the original kernel. The boundary condition of each case denotes the dimension of the corresponding sub-kernel (notice the different floor and ceiling functions in each). Hence, decomposing a 3×3 kernel results in four sub-kernels of shapes 2×2, 1×2, 2×1, and 1×1, confirming the specific example above.

5.3.2 Exploiting Inter-Layer Activation Reuse

A beneficial trait of our transformation is that each sub-convolution reads the same *ifmap*, which in modern DNNs does not fit in on-chip buffers and must spill to main memory. In contrast, our transformation can uniquely exploit *inter-layer activation reuse* because each sub-convolution layer shares the same *ifmap*. The challenge is to systematically maximize the reuse exploited across the entire network while minimizing the inference latency.

We primarily consider *loop tiling*, which is known to be critical to exploiting data reuse in DNNs (Mullapudi et al., 2016; Yang et al., 2018). Prior work in DNN tiling predominately searches for the tiling strategy in a brute-force manner (Ma et al., 2017; Hegde et al., 2018). However, brute-force search does not scale to stereo DNNs for two reasons. First, our translation scheme significantly increases the number of layers, each of which must be individually searched. For instance in the example of Fig. 5.3, the number of layers quadruples; a 3D kernel could increase layers by $8\times$. Second, exploiting the inter-layer *ifmap* reuse adds another scheduling dimension, further increasing the search space.

Instead of a search, we formulate the reuse optimization as a constrained optimization problem, minimizing layer latency while satisfying hardware resource constraints. Our optimization can be efficiently solved using a greedy algorithm.

Architectural Assumptions We first describe the underlying architecture that the optimization formulation assumes. Overall, we make standard assumptions that generally hold across the vast majority of current DNN accelerators. Sec. 5.4.2 describes the hardware architecture in detail.

We assume a systolic array accelerator. Each Processing Element (PE) performs one MAC operation per cycle (Jouppi et al., 2017b; Samajdar et al., 2018). Systolic arrays use a very efficient neighbor-to-neighbor communication mechanism, particularly well suited to convolution. Alternatively, our formulation could also be extended to support spatial arrays (Chen et al., 2016), which offer more flexible control at higher hardware cost.

We assume that the accelerator has a unified on-chip buffer (scratchpad) for the *ifmap*, kernels, and *ofmap*. This buffer is generally too small to hold all the data for a whole layer. Therefore, the *ofmap* is computed in multiple rounds. Only part of the *ifmap* and the kernels are stored in the buffer each round. The optimal scheduling of partial *ifmap* and kernels in the buffer for each round is critical to maximizing reuse.

The buffer is evenly split into working and filling sections for double-buffering.

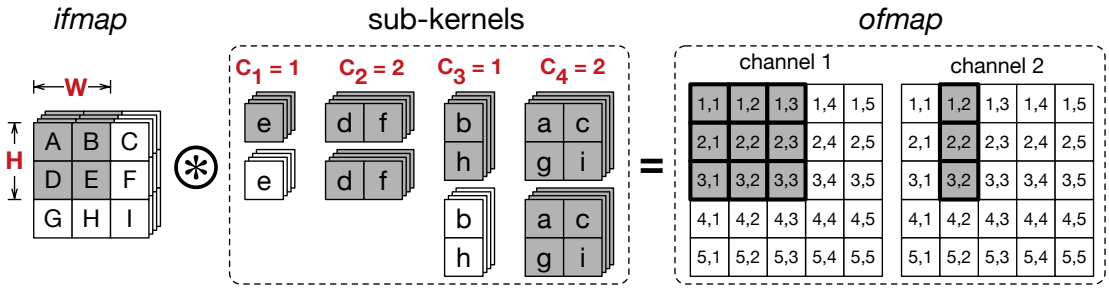


Fig. 5.4: Tiling in a translated deconvolution with a 3×3 kernel split into four sub-kernels. With a tiling strategy $W = 2, H = 2, C_1 = 1, C_2 = 2, C_3 = 1, C_4 = 1$, only the shaded elements are loaded into the buffer. The *ofmap* elements generated in this round (shaded) are also stored in the buffer.

While the PE array is computing the current round using data in the working buffer, the data for the next round is pre-fetched to the filling buffer. The next round starts only when the filling buffer is full. This design choice guarantees that any data access by the PEs will hit in the buffer without stalling the PE array.

Optimization Formulation We follow a layer-wise execution model, in which a layer only starts after the previous layer finishes. Therefore, minimizing the total latency is equivalent to minimizing the latency of each individual layer. We describe how the latency of a deconvolution layer is formulated and optimized. Our formulation can be easily extended to support a convolution layer, which can be regarded as a special case of deconvolution without `ILAR`.

The optimization objective is to minimize the deconvolution layer's latency given hardware resource constraints. Note that since a deconvolution is translated to a set of convolutions, it is the cumulative latency of these sub-convolutions that is of interest. The optimization problem is formulated as follows:

$$\min \; L(\Theta, \phi) \tag{5.3}$$

$$s.t. \; R(\Theta) \leq R^* \tag{5.4}$$

where Θ denotes a particular hardware configuration, and $R(\cdot)$ is the configuration's

hardware resources, which must not exceed the specified resource budget R^*. We consider three main types of hardware resources: 1) PE array size, 2) on-chip buffer size, and 3) off-chip memory bandwidth.

Latency $L(\cdot)$ is affected by both the hardware configuration (Θ) and the tiling schedule (ϕ). The optimal tiling is determined by the following variables: 1) the dimension of the *ifmap* tile to be loaded into the buffer (W and H), and 2) the number of filters in each **sub-kernel** k to be loaded into the buffer (C_k). Critically, C_k can be different for each sub-kernel. Fig. 5.4 illustrates these optimization variables, with an example where part of the *ifmap* is convolved with certain filters of the four sub-kernels to generate a partial *ofmap*. The vector $\overrightarrow{C}$ denotes the collection of all C_k.

With double buffering, a layer L's latency is the cumulative latency across all N rounds. The latency of each round (l^i) is determined by the maximum value between the memory access time (l^i_m) and the compute time (l^i_c) of the round:

$$L(\Theta, \phi) = \sum_{i=1}^{N} l^i(\Theta, \phi), \qquad l^i(\Theta, \phi) = \max(l^i_c, l^i_m) \tag{5.5}$$

With double-buffering, l^i_c is determined by two sets of parameters: 1) W^i, H^i, and $\overrightarrow{C^i}$, which decide the total compute demand, and 2) the PE array size, A^*, which decides the compute capability. l^i_c is the cumulative latency of processing each individual sub-kernel:

$$l^i_c = \sum_{k=1}^{|\overrightarrow{C^i}|} \left\lceil \frac{W^i_k \times H^i_k \times I \times C^i_k \times H^i \times W^i}{A^*} \right\rceil \tag{5.6}$$

where $|\overrightarrow{C^i}|$ denotes the total number of sub-kernels in round i, W^i and H^i are the dimensions of the *ifmap* tile loaded into the buffer in round i, W^i_k and H^i_k are the dimensions of sub-kernel k in round i[1], C^i_k denotes the number of filters in sub-kernel k loaded into the buffer in round i, and I is the number of input channels. The ceil operator indicates

[1]The sub-kernels' dimensions do not change across rounds. Given a k, W^i_k and H^i_k are constants for any i. For the consistency of the notations, we still use W^i_k and H^i_k.

that the next sub-kernel can not start until the previous sub-kernel is finished even if the PE array is under-utilized. This is because only one sub-kernel can be calculated on the systolic array at a time as sub-kernels vary in their shapes.

The memory access time, l_m^i, is determined by the available memory bandwidth, B^*, and the amount of data that needs to be transferred to/from DRAM each round, which in turn depends on the reuse order: whether the *ifmap* tile or the sub-kernels remain in the buffer across consecutive rounds. A binary variable β denotes this reuse order, and l_m^i becomes:

$$l_m^i = \beta \times l_{m:W}^i + (1-\beta) \times l_{m:In}^i,\ \beta \in \{0,1\} \tag{5.7}$$

where $l_{m:In}^i$ is the memory access latency if the *ifmap* remains in the buffer, and $l_{m:W}^i$ denotes the memory latency if the sub-kernels remain in the buffer. Specifically:

$$l_{m:W}^i = \ (\Delta IF^i + \sum_{k=1}^{|\overrightarrow{C^i}|} \Delta OF_k^i) \times \frac{1}{B^*} \tag{5.8}$$

$$l_{m:In}^i = \ \sum_{k=1}^{|\overrightarrow{C^i}|} (\Delta W_k^i + \Delta OF_k^i) \times \frac{1}{B^*} \tag{5.9}$$

where the terms with prefix Δ denote the amount of data that needs to be loaded from DRAM. Depending on the reuse order, either the *ifmap* elements (ΔIF^i), or the sub-kernels (ΔW_k^i) are loaded. The newly computed *ofmap* elements (ΔOF_k^i) are always stored back to DRAM. Note that ΔW_k^i, ΔIF^i, and ΔOF_k^i are all deterministic functions of W_k^i, H_k^i, W^i, H^i, and $|\overrightarrow{C^i}|$. The expression of ΔW_k^i, ΔIF^i, and ΔOF_k^i are determined by W_k^i, H_k^i, and $|\overrightarrow{C^i}|$. Their exact expressions are shown below:

$$\Delta W_k^i = W_k^i \times H_k^i \times C_k^i$$

where C_k^i denotes the total number of sub-kernel k in round i; W_k^i and H_k^i are the dimensions of sub-kernel k in round i.

$$\Delta IF^i = W^i \times H^i \times I$$

where W^i and H^i are the weight and height of an *ifmap* tile to be loaded in round i, and I is the number of input channels.

$$\Delta OF_k^i = \frac{W^i \times H^i \times C_k^i}{s^2}$$

where s denotes the stride of this layer. The on-chip buffer capacity (Buf^*) imposes the constraint:

$$\Delta IF^i + \sum_{k=0}^{|\overrightarrow{C^i}|}(\Delta OF_k^i + \Delta W_k^i) \leq Buf^* \tag{5.10}$$

Finally, C_k^i and N must satisfy:

$$\forall k \in \{1, 2, ..., |\overrightarrow{C}|\}, \mathbb{C} = \sum_{i=1}^{N} C_k^i \tag{5.11}$$

where $\mathbb{C}$ denotes the number of output channels of a layer, which is a constant invariant to k and i.

Overall, this formulation minimizes the latency L with respect to W^i, H^i, and C_k^i ($i \in \{1, 2, ..., N\}$, $k \in \{1, 2, ..., |\overrightarrow{C}|\}$), under the hardware resource constraints A^*, B^*, and Buf^*.

Efficient Solver The above constrained-optimization problem has a non-convex objective and constraints, and thus has no closed-form solutions. To derive a solution efficiently, we convert this problem to a Knapsack-like structure, where each filter in each sub-kernel is an *item*, the size of each filter is the *weight*, and the number of MAC operations associated with each filter is the *value*.

To solve the Knapsack problem, we use a simple greedy heuristic that prioritizes filters from large sub-kernels with standard dynamic programming. In contrast to the classic 0/1 Knapsack problem, our problem formulation requires us to consume all the items, since all the filters in each sub-kernel are required to finish a convolution. We therefore iteratively apply the greedy solver until all the items are used. The solver is executed offline and finishes within one second on an Intel Core i5-7500 CPU.

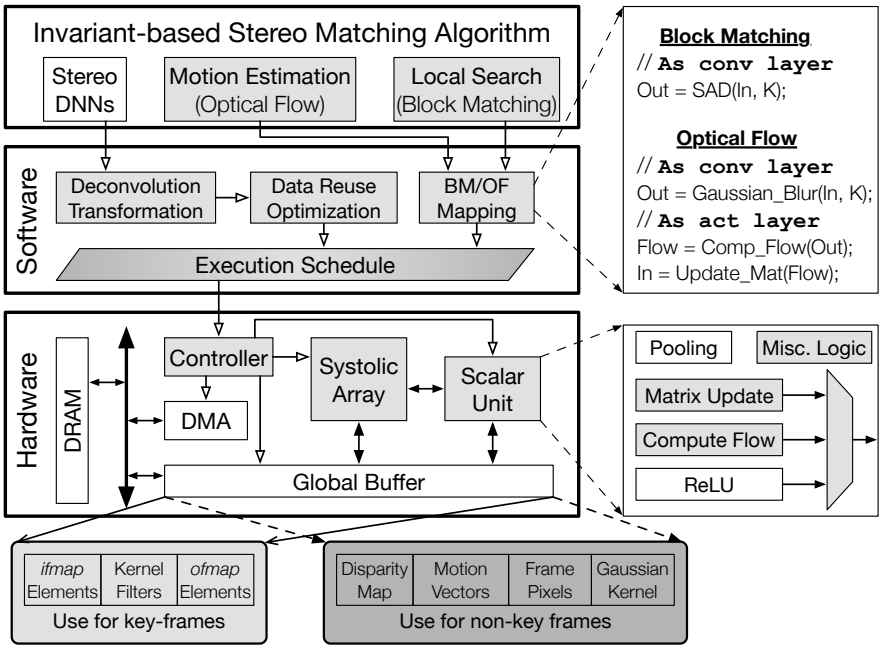


Fig. 5.5: The ASV overview with augmentations shaded.

5.4 The ASV System

Building on top of the ISM algorithm and the deconvolution optimizations, this section presents the software and hardware system of ASV. Fig. 5.5 gives a high-level system overview of ASV. We first present the software system (Sec. 5.4.1), and then discuss the architecture design decisions (Sec. 5.4.2).

5.4.1 Software System

The goal of the software system in ASV is to map the ISM algorithm to the underlying hardware. The static mapping is done offline. There are three components in the ISM algorithm to map: stereo matching DNN, motion estimation, and local correspondence search. We rely on the user to supply a particular stereo DNN depending on their accuracy needs. Motion estimation and correspondence search are implemented using optical flow (OF) and block matching (BM), respectively, as described in Sec. 5.2.3.

We now describe how each component is processed by the software.

Mapping Stereo Matching DNN For the deconvolution layer, the ASV software performs the deconvolution transformation, as well as the data reuse optimization. For convolution layers, while the deconvolution transformation does not apply, we apply the data reuse optimization *without* ILAR. In the end, we obtain a transformed stereo DNN along with an execution schedule, which are both consumed by the hardware at runtime. The schedule includes the tiling strategy and buffer partitioning strategy for each layer.

Mapping OF/BM The software maps the OF and BM algorithms in ISM to a set of convolution and/or activation operations that are directly interfaced with conventional DNN accelerators. The software translates the BM operation to a convolution layer (Sec. 5.2.3), but calculates SAD instead of dot product at each window.

The OF computations include Gaussian blur, "Compute Flow" and "Matrix Update" operations (Sec. 5.2.3), shown in the top-right box in Fig. 5.5. Gaussian blur is naturally expressed as a convolution layer with one output channel. "Compute Flow" and "Matrix Update" are point-wise operations expressed as special activation functions.

5.4.2 Hardware Architecture

Leveraging the software pass, the hardware requires only minimal, structured augmentations on top of a conventional DNN accelerator. We start from a baseline DNN accelerator and describe how the compute, memory, and control logic is augmented with ASV-specific architectural extensions.

Compute Our baseline DNN accelerator consists of a TPU-like systolic PE array for convolution and a scalar unit for non-convolution operations, e.g., activation (Jouppi et al., 2017b). Each PE consists of two 16-bit input registers, a 16-bit fixed-point MAC unit with a 32-bit accumulator register, and simple trivial control logic. This is identical to the PE in the TPU (Jouppi et al., 2017b).

We use the systolic array as the baseline due to its efficiency in handling convolutions. However, our software optimizations do not depend on a particular baseline DNN architecture. Alternatives such as more flexible spatial architectures (Chen et al., 2014a, 2016) are also suitable, albeit requiring different constrained-optimization formulations to those presented in Sec. 5.3.2. We will later demonstrate the effectiveness of our deconvolution optimizations on Eyeriss (Chen et al., 2016).

ASV augments both the systolic array and the scalar unit in the baseline architecture to support the ISM algorithm. First, each PE is extended with the capability to accumulate absolute differences (i.e., $a \leftarrow a + |b - c|$) in addition to MAC in order to support BM. Second, we extend the scalar unit to support two additional point-wise operations: "Compute Flow" and "Matrix Update"; both are required by OF (as illustrated in the bottom-right box in Fig. 5.5).

Finally, the hardware includes a very small amount of additional logic to support the remaining operations in the ISM algorithm that are inefficient to map to either the systolic array or the point-wise scalar unit. These operations are comparisons and control-flow, and are orders of magnitude less costly in area and power compared to the systolic array and the scalar unit. For instance, BM requires comparing the SAD values across different matched blocks, and OF requires checking the value boundaries during "Matrix Update".

Memory ASV uses the familiar three-level memory hierarchy (Li et al., 2019). Each PE has a small register file to exploit intra/inter-PE data reuse. A DMA engine coordinates data transfer between the on-chip global buffer and off-chip memory. The global buffer is temporally shared between key frames and non-key frames. When processing key frames, the global buffer holds the ifmap, kernels, and ofmaps. The exact buffer partitioning is dictated by the ASV software.

When processing non-key frames, the global buffer holds four pieces of data: the pixels of the current and key frames, the Gaussian kernel, the motion vectors, and the disparity maps. The frame pixels dominate the storage requirement but could be

tiled because they are used in Gaussian blur and BM, both of which are convolution operations. The rest of the data cannot be tiled, and thus imposes a minimum buffer size. Assuming qHD resolution (960×540), we enforce a minimum buffer size of about 512 KB.

Control A micro-sequencer is used to coordinate the computation and memory accesses. In ASV, the sequencer also chooses key frames. Although complex adaptive schemes are feasible (Zhu et al., 2018b; Buckler et al., 2018), we found that a simple strategy to statically set the key-frame window suffices (Sec. 5.6.2).

5.5 Evaluation Methodology

This section introduces the basic hardware and software setup (Sec. 5.5.1), and outlines the evaluation plan (Sec. 5.5.2).

5.5.1 Basic Setup

Hardware Implementation We develop validated RTL implementations for ASV hardware. The hardware is based on a systolic array architecture, consisting of 24×24 PEs clocked at 1 GHz. Each PE is capable of performing both the MAC and absolute difference operations. The hardware also has a scalar unit clocked at 250 MHz, which consists of 8 parallel lanes, each capable of performing the ReLU activation function as well as the point-wise matrix update and compute flow operations required by OF. The on-chip buffer (SRAM) is 1.5 MB in size and is banked at a 128 KB granularity. While we primarily evaluate ASV using this configuration, we will later show the sensitivity of ASV performance to different hardware resource configurations.

The RTL is implemented using Synposys synthesis and Cadence layout tools in TSMC 16nm FinFET technology, with SRAMs generated by an ARM compiler. Power is simulated using Synopsys PrimeTimePX, with full annotated switching activity. The

off-chip DRAM is modeled after four Micron 16 Gb LPDDR3-1600 channels (Micron, 2014). Overall, the accelerator layout has a total area of $3.0\,\mathrm{mm}^2$, and produces a raw throughput of 1.152 Tera operations per second.

Stereo DNNs The ISM algorithm can use an arbitrary stereo DNN. We evaluate four state-of-the-art DNNs: FLOWNETC (Fischer et al., 2015), DISPNET (Mayer et al., 2016), GC-NET (Smolyanskiy et al., 2018), and PSMNET (Chang and Chen, 2018).

Dataset We evaluate ASV on two widely-used datasets: SceneFlow (Mayer et al., 2016) and KITTI (Menze and Geiger, 2015). SceneFlow contains 26 pairs of synthetic stereo videos to mimic various scenarios with different depth ranges. KITTI contains 200 pairs of stereo frames captured from real street views that cover varying driving scenarios and conditions.

We use the standard "three-pixel-error" accuracy metric (Geiger, 2012; Menze and Geiger, 2015), which considers a pixel's depth to be correct if its disparity error is less than 3 pixels compared to ground truth. We then report the percentage of correct pixels, following the convention in the vision and robotics literature (Mayer et al., 2016; Kendall et al., 2017; Chang and Chen, 2018; Smolyanskiy et al., 2018).

5.5.2 Evaluation Plan

Our goal is to demonstrate the effectiveness of ASV over generic CNN accelerators that are not optimized for stereo vision workloads. We separate the efficiency gains of the new ISM algorithm from that of the deconvolution optimizations.

Baselines Our baseline is a generic systolic array CNN accelerator, which executes stereo DNNs without any ASV optimizations. Today's CNN accelerators mostly statically partition the on-chip buffer across *ifmap*, weights, and *ofmap*. To obtain a strong baseline, we determine the partitioning strategy by exhaustively searching all the partitions offline and use the one that achieves the lowest latency for the entire DNN. Note

that the same partition is used for all the layers whereas our data reuse optimization generates different partitions for different layers.

We also compare against Eyeriss (Chen et al., 2016), a DNN accelerator based on a more flexible spatial architecture. Eyeriss performance and energy are obtained using the public simulator (Gao et al., 2017). For a fair comparison, we configure Eyeriss to have the same PE counts, on-chip memory capacity, and memory bandwidth as ASV. Finally, to establish a baseline, we also show the results of the Pascal mobile GPU found in the 16 nm Nvidia Parker SoC hosted on the Jetson TX2 development board (Nvidia, 2017c). We use the built-in power sensing circuitry to obtain energy consumption.

ASV Variants We present an ablation study on ASV to separate the gains from different optimizations:

- ISM: ISM algorithm without deconvolution optimizations.
- DCO: Deconvolution optimizations without ISM algorithm.
- ISM+DCO: Both ISM and deconvolution optimizations.

5.6 Evaluation

We first show that ASV adds negligible overhead to the baseline DNN accelerator (Sec. 5.6.1) and introduces negligible accuracy loss (Sec. 5.6.2). We then show the performance and energy improvements of ASV (Sec. 5.6.3), which are robust against the underlying hardware configuration (Sec. 5.6.4).

5.6.1 Hardware Overhead

Owing to the software transformations, ASV only minimally augments existing DNN accelerators. Relative to the baseline accelerator, ASV extends each PE to support

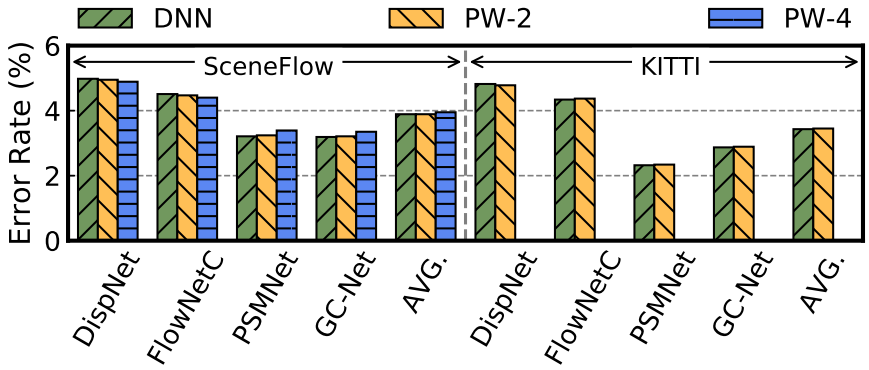


Fig. 5.6: Error rate comparison between the ISM algorithm in ASV and the DNN baselines.

accumulating absolute difference. This adds 6.3% area ($15.3\,\mu m^2$) and 2.3% power ($0.02\,mW$) overhead *per PE*. ASV also extends the scalar unit to support new point-wise operations, with an area and power overhead of $2\,mm^2$ and $2.2\,mW$, respectively. The overall area and power overhead introduced by ASV are both below 0.5%.

5.6.2 Accuracy Results

ASV matches or even outperforms DNN accuracy. Fig. 5.6 shows the accuracy of applying the ISM algorithm to stereo matching DNNs. We use *Propagation Window* (PW) to denote how far in time the correspondence invariant is propagated, which in turn decides how often key frames are selected. With PW-2, every other frame is selected as a key frame, and for PW-4, every fourth frame is a key frame. Note that the KITTI dataset contains at most two consecutive frames, and thus we evaluate only PW-2.

On both datasets, PW-2 retains the same accuracy as the stereo DNNs. On SceneFlow, PW-4 results in only 0.02% accuracy loss. In some cases, ISM combined with the DNNs can outperform the DNNs alone. For instance, applying the ISM algorithm with FLOWNETC reduces error by 0.11% at PW-4. Overall, our experiments shows that by leveraging the correspondence invariant over time, ISM is able to preserve the DNN-like accuracy with cheap, classic stereo matching algorithms. We will now show

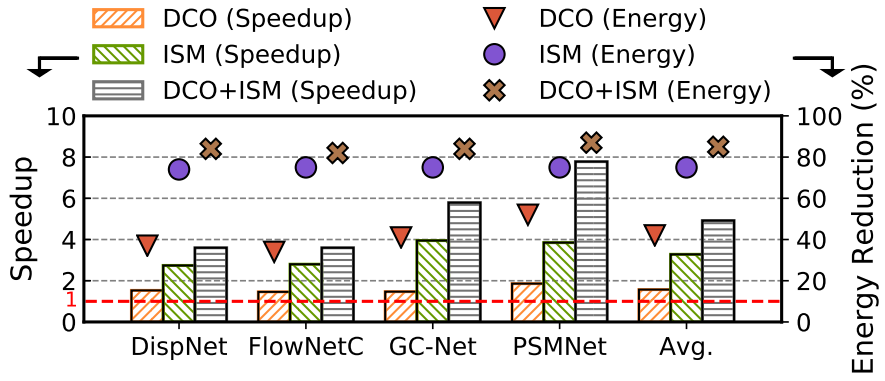


Fig. 5.7: The speedup and energy reduction of the three ASV variants over the baseline.

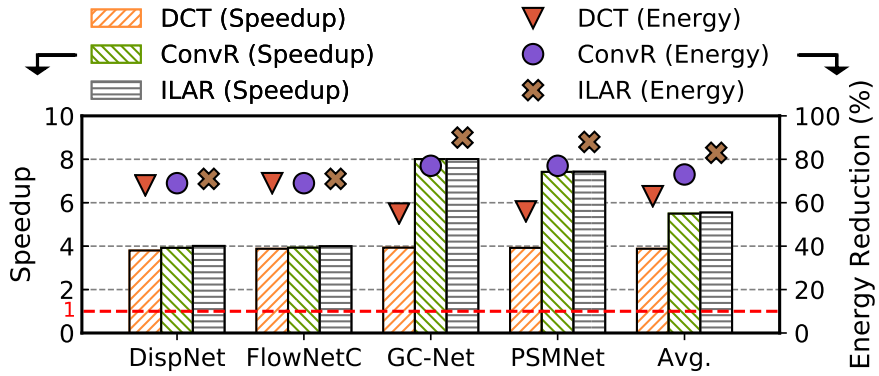
that ASV achieves high accuracy while greatly improving the performance and energy-efficiency of stereo vision.

5.6.3 Speedup and Energy Reduction

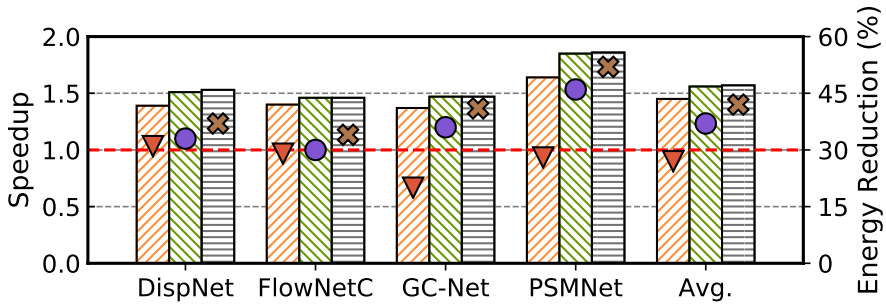
ASV significantly improves the performance and energy consumption of stereo vision. To understand the contributions of the ISM algorithm and the deconvolution optimizations, Fig. 5.7 shows the speedup and energy reduction of the three ASV variants (Sec. 5.5.2) over the baseline when applied to different stereo DNNs. We choose PW-4 for the ISM algorithm. On average, combining ISM and deconvolution optimizations (DCO) ASV achieves 4.9× speedup and 85% energy reduction. Specifically, ISM achieves, on average, 3.3× speedup and 75% energy reduction, while DCO achieves 57% performance improvement and 38% energy reduction. ISM contributes more than DCO because ISM avoids DNNs in non-key frames altogether by using the much cheaper BM and OF algorithms (Sec. 5.2.3).

Next, we dissect different optimization components within DCO to further understand the effect of each optimization.

Deconvolution Optimizations Deconvolution optimizations consist of two components: the deconvolution to convolution transformation (`DCT` - Sec. 5.3.1) and the data-reuse optimization (Sec. 5.3.2). In particular, our data-reuse formulation unifies



(a) Speedup and energy reduction on deconvolution layers only.



(b) Speedup and energy reduction on the entire network.

Fig. 5.8: The speedup and energy reduction of various deconvolution optimizations. Higher is better.

the exploitation of two kinds of reuse: the conventional data reuse in convolution layers and the inter-layer activation reuse in deconvolutions that is uniquely exposed by `DCT`. To clearly tease apart our contributions, we show the results of both the conventional reuse optimization (`ConvR`), which is obtained by applying our reuse optimizer (Sec. 5.3.2) *without* exploiting inter-layer activation reuse, and the additional effect of exploiting inter-layer activation reuse (`ILAR`).

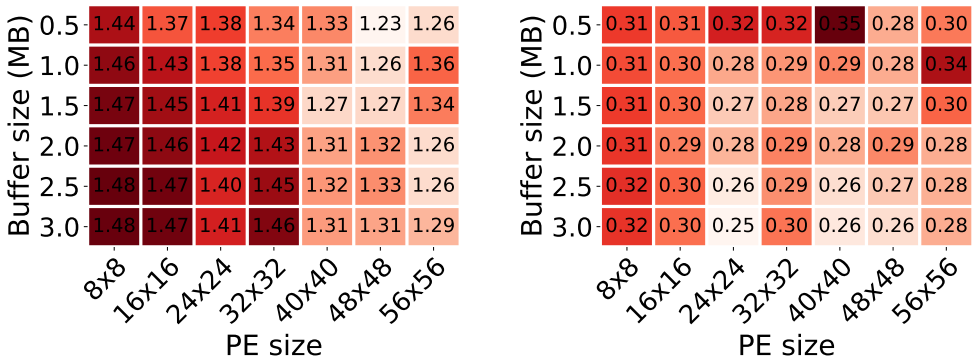
Fig. 5.8 shows the speedup and energy reduction of `DCT`, `ConvR`, and `ILAR`. Fig. 5.8a shows the improvements of deconvolution layers only, and Fig. 5.8b shows the improvements of the entire network. The majority of speedup is from deconvolution transformation, which yields an average 3.9× speedup on deconvolution layers alone and 1.4× speedup on the entire network. On top of `DCT`, `ConvR` and

ILAR further increase speedup to 5.6× and 1.6× on deconvolution layers alone and the entire networks, respectively.

Across different stereo DNNs, we find that 3D DNNs (GC-NET and PSMNET) have a speedup of 7.7× on deconvolution layers, higher than the 3.9× speedup of 2D DNNs (DISPNET and FLOWNETC). The reason is twofold. First, 3D DNNs have the higher percentage of zero-padding than 2D DNNs (8× vs. 4×), which are effectively eliminated by our deconvolution transformation. Second, after the deconvolution transformation the 3D DNNs have many small kernels (e.g., 1×1×1), which leads to low data-reuse. Thus, reuse optimizations become more critical to these networks. In contrast, most 2D stereo DNNs inherently have better data reuse with larger kernels (e.g., 5×5). We also observe that ConvR and ILAR have similar performance. This is because both optimize the data reuse to the extent that the layer processing becomes limited by the PE size.

While ILAR is similar in speedup compared to ConvR, ILAR is much more effective in reducing energy than ConvR. To demonstrate this, Fig. 5.8 overlays the energy reductions of different DCO variants on the right y-axis. DCO achieves 83% energy reduction on deconvolution alone and 38% on the entire network. Specifically, DCT reduces the deconvolution energy by 62%; ConvR and ILAR further improve the energy reduction to 73% and 83%, respectively.

The energy saving of DCT comes from eliminating redundant movement of padded zeros in the upsampled *ifmap*. ILAR achieves additional energy reduction over ConvR by exploiting inter-layer activation reuse, a unique reuse behavior in our transformed deconvolution layers. 3D DNNs benefit much more from ILAR than 2D DNNs, as is evident by examining the additional energy reductions of ILAR over ConvR across networks. This is because 3D stereo DNNs have low *ifmap* data-reuse; ILAR uniquely exploits inter-layer *ifmap* reuse, and thus reduces more memory traffics.



(a) Speedup.

(b) Energy reduction.

Fig. 5.9: Sensitivity analysis of DCO speedup and energy reduction with buffer size and PE array size on FLOWNETC. Speedup is normalized to the corresponding configurations, not to a single, common baseline.

5.6.4 Sensitivity Analysis

While the speedup and energy reduction studied so far are based on one representative baseline accelerator configuration (Sec. 5.5.1), we find that our deconvolution optimization generally achieves similar improvements on other hardware configurations with resource provisions. In particular, we focus on two key types of hardware resources: PE size and on-chip buffer size. For brevity, we only report results on FLOWNETC (Fischer et al., 2015), but the trends generally hold.

Fig. 5.9a and Fig. 5.9b show how DCO's average speedup and energy reduction of the entire network vary with different PE size and buffer size combinations, respectively. Note that the results are normalized to their corresponding hardware configurations rather than to the baseline described in Sec. 5.5.1. For instance, on the hardware with an 8×8 PE array and a 0.5 MB on-chip buffer, DCO achieves an 1.44× speedup.

DCO achieves speedups of 1.2× – 1.5× and energy reductions of 25% – 35% across different hardware capabilities, demonstrating broad applicability. In general, the performance improvement of DCO is more pronounced with small PE arrays, where the

performance is compute-bound. As the PE size increases, the performance becomes memory bound, such that memory bandwidth limitations mask the benefit of data reuse. In addition, as the buffer size increases, the reuse opportunities inherently exposed by the buffer is higher, and thus data reuse optimizations become less critical, hence the lower energy savings.

5.7 Related Work

Stereo Vision Accelerators Recently, commercial mobile vision systems (Zhu et al., 2018a) have started integrating dedicated stereo accelerators, such as the Stereo Depth Block in the Movidius Enhanced Vision Accelerator Suite (Intel, 2017), and the Stereo & Optical Flow Engine (SOFE) in the Nvidia Xavier mobile SoC (Nvidia, 2018). From publicly available details, these are fixed-functioned accelerators targeting classic stereo algorithms, similar to previous stereo vision accelerators (Gudis et al., 2013; Ttofis and Theocharides, 2014; Yang, 2014; Wang et al., 2015; Mazumdar et al., 2017). In contrast, ASV combines the efficiency of classic stereo algorithms with the accuracy of stereo DNNs.

Motion-based Algorithms Our ISM algorithm shares a similar key observation as some recent motion-based vision algorithms such as EVA^2 (Buckler et al., 2018) and Euphrates (Zhu et al., 2018b), in that correlation across frames in a video stream can be used to simplify continuous vision tasks. Euphrates (Zhu et al., 2018b) focuses on computing regions-of-interest (ROIs) in object detection and tracking tasks. In contrast, stereo vision is concerned with the depth of the whole frame rather than discrete ROIs. EVA^2 (Buckler et al., 2018) is not limited to ROIs. However, it relies on estimating the motion of an intermediate activation's receptive field. In the stereo task, the receptive field of an intermediate activation necessarily spans both the left and right images. Thus, the motion of the receptive field would be difficult, if not impossible, to calculate.

Fundamentally, motion-based relaxations fall within the realm of incremental com-

puting, a general technique used in program analysis and optimization (Michie, 1968; Pugh and Teitelbaum, 1989) and applies beyond the temporal and/or vision domain. Diffy (Mahmoud et al., 2018) exploits the spatial similarity across pixels in the same frame to improve DNN efficiency. Riera et al. (Riera et al., 2018) exploit repeated *ifmap* elements in speech recognition.

Deconvolution Sparsity Many prior studies optimize hardware to exploit sparsity in DNNs (Wen et al., 2016; Albericio et al., 2016; Han et al., 2016b; He et al., 2017b; Kung et al., 2019; Wu et al., 2018b; Whatmough et al., 2018; Lee et al., 2019). Stereo vision DNNs make use of deconvolution layers, which expose structured sparsity patterns. Recent work has prosed specialized hardware specifically for exploiting sparsity in deconvolution layers (Yazdanbakhsh et al., 2018; Song et al., 2018). Our observation, however, is that mitigating sparsity-induced efficiencies in deconvolution does not necessarily require hardware support. We propose novel software optimizations to eliminate the compute inefficiencies without hardware changes.

Data-Reuse Optimization Exploiting data-reuse (through tiling) is critical to DNN efficiency (Chen et al., 2014b, 2016; Alwani et al., 2016; Mullapudi et al., 2016; Yang et al., 2016; Gao et al., 2017; Ma et al., 2017; Hegde et al., 2018; Whatmough et al., 2019; Kwon et al., 2018; Gao et al., 2019; Whatmough et al., 2019a). Orthogonal to generic data-reuse, we identify a new reuse dimension, inter-layer activation reuse (`ILAR`), that is uniquely enabled by our deconvolution transformation.

Previous DNN mapping frameworks mostly rely on exhaustive search (Yang et al., 2016, 2018; Hegde et al., 2018), which does not scale to exploiting `ILAR` (Sec. 5.3.2). TETRIS (Gao et al., 2017) also uses a constrained optimization for DNN scheduling, albeit with certain problem-specific simplifications. However, it does not exploit `ILAR`. ASV directly optimizes for latency rather than memory traffic (Yang et al., 2016; Gao et al., 2017) or resource utilization (Ma et al., 2017).

6 Point Cloud Vision System

Similar to images, point clouds are also widely used in many vision computing systems, such as drone/robotic navigation, and AR/VR. This chapter introduces the basic concepts of point cloud and its vision applications in Sec. 6.1. Then, the general computation flow of today's point cloud algorithms with the common computation inefficiencies is discussed in Sec. 6.2. Lastly, Sec. 6.3 highlights the memory inefficiencies that currently exist in point cloud analytics.

6.1 Introduction

Point Cloud A point cloud is an unordered set of points in the 3D Cartesian space. Each point is uniquely identified by its $\langle x,\ y,\ z\rangle$ coordinates. While point cloud has long been used as a fundamental visual data representation in fields such as 3D modeling (Alliez et al., 2017) and graphics rendering (Levoy and Whitted, 1985; Rusinkiewicz and Levoy, 2000; Pfister et al., 2000; Gross and Pfister, 2011), it has recently received lots of attention in a range of emerging intelligent systems such as autonomous vehicles (Geiger et al., 2012a), robotics (Whitty et al., 2010), and AR/VR devices (Stets et al., 2017).

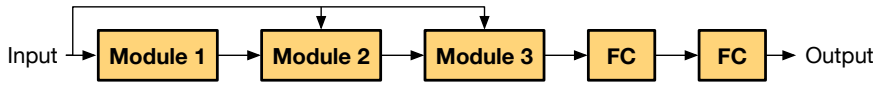
Point Cloud Acquisitions There are numerous ways to obtain point clouds. The

two major methods are laser scanning and photogrammetry (Lehtola et al., 2017). In laser scanning, a technology called LiDAR which stands for Light Detection and Ranging is commonly used to harvest point clouds by scanning over a scene or an object. On the other hand, photogrammetry relies on photos and performs 3D reconstruction software to create point cloud representations. Although point clouds generated from photogrammetry are often more accurate due to a large number of individual points created from a 3D reconstruction, LiDAR is able to capture a large space in a real-time manner. Therefore, LiDAR is often preferred for many real-time vision applications, e.g. autonomous driving and drone navigation. For these applications, LiDAR becomes an essential sensor in a part of their perception systems.

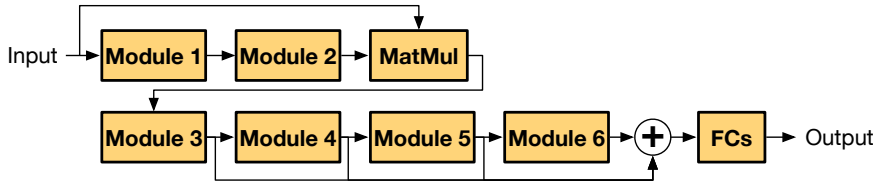
For their perception systems, LiDARs generate massive amounts of point cloud data. For instance, the Velodyne HDL64E LiDAR generates hundreds of thousands of points in each frame, amounting to up to 26 MB of raw data per second. Effectively compressing point clouds in real-time enables autonomous machines to be closely connected with each other and with the cloud, ushering in a new era in distributed and cloud robotics. For instance, efficient point cloud compression would enable offloading compute-intensive perception tasks (e.g., object detection) to the cloud to reduce the perception latency; similarly, collaborative decision-making across robots relies on efficient point cloud compression to exchange information.

Point Cloud Analytics Similar to conventional visual analytics that analyzes images and videos, point cloud analytics distill semantics information from point clouds. Examples include object detection (Geiger et al., 2012a), semantics segmentation (Behley et al., 2019), and classification (Wu et al., 2015). While image and video analytics have been well-optimized, point cloud analytics require different algorithms and are much less optimized.

Point cloud algorithms operate by iteratively extracting features of each point. Conventional point cloud algorithms use "hand-crafted" features such as FPFH (Rusu et al., 2009) and SHOT (Tombari et al., 2010). Recent deep learning-based algorithms use



(a) Network architecture of PointNet++ (Qi et al., 2017b).



(b) DGCNN (Wang et al., 2019a) network architecture. "+" is tensor concatenation.

Fig. 6.1: Point cloud networks consist of a set of modules, which extract local features from the input point cloud iteratively and hierarchically to calculate the final output.

learned features and have generally out-performed conventional algorithms (Chen et al., 2016). This paper thus focuses on deep learning-based algorithms.

This dissertation focuses on deep learning-based algorithms that directly manipulate raw point clouds. Other data representations such as 2D projections of 3D points and voxelization suffer from low accuracy and/or consume excessively high memory (Liu et al., 2019c).

6.2 Computation in Point Cloud Analytics

In this section, Sec. 6.2.1 first introduces the general flow of point cloud algorithms and identifies key operators. Sec. 6.2.2 then characterizes point cloud algorithms on today's hardware systems to understand the algorithmic and compute inefficiencies in point cloud analytics.

6.2.1 Point Cloud Network Architecture

Module The key component in point cloud algorithms is a *module*. Each module transforms an input point cloud to an output point cloud, similar to how a convolu-

tion *layer* transforms an input feature map to an output feature map in conventional CNNs. A point cloud network assembles different modules along with other common primitives such as fully-connected (FC) layers. Fig. 6.1a and Fig. 6.1b illustrate the architecture of two representative point cloud networks, PointNet++ (Qi et al., 2017b) and DGCNN (Wang et al., 2019a), respectively.

Module Internals Each point $\mathbf{p}$ in a point cloud is represented by a feature vector, which in the original point cloud is simply the 3D coordinates of the point. The input point cloud to a module is represented by an $N_{in} \times M_{in}$ matrix, where N_{in} denotes the number of input points and M_{in} denotes the input feature dimension. Similarly, the output point cloud is represented by an $N_{out} \times M_{out}$ matrix, where N_{out} denotes the number of output points and M_{out} denotes the output feature dimension. Note that N_{in} and N_{out} need not be the same; neither do M_{in} and M_{out}.

Internally, each module extracts local features from the input point cloud. This is achieved by iteratively operating on a small *neighborhood* of input points, similar to how a convolution layer extracts local features of the input image through a sliding window. Fig. 6.2 illustrates this analogy.

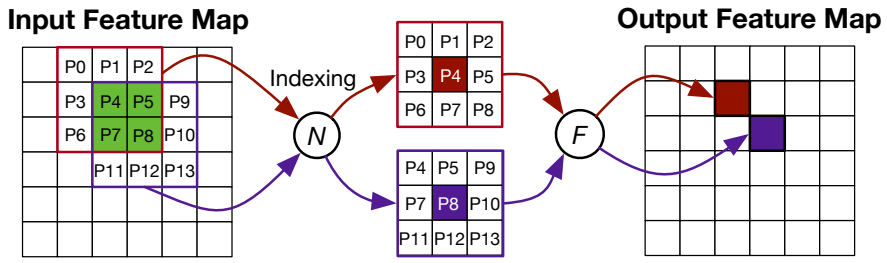
Specifically, each output point $\mathbf{p}_o$ is computed from an input point $\mathbf{p}_i$ in three steps — neighbor search ($\mathcal{N}$), aggregation ($\mathcal{A}$), and feature computation ($\mathcal{F}$):

$$\mathbf{p}_o = \mathcal{F}(\mathcal{A}(\mathcal{N}(\mathbf{p}_i),\ \ \mathbf{p}_i)) \tag{6.1}$$

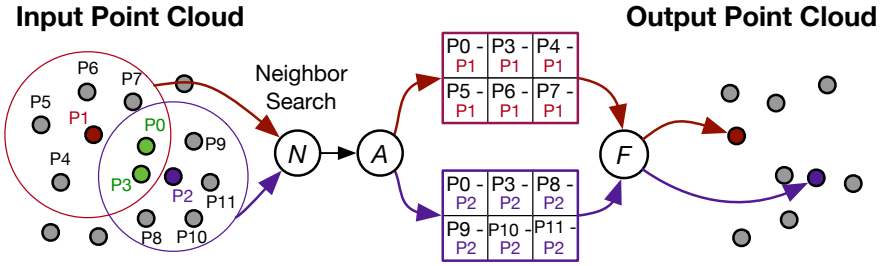
where $\mathcal{N}$ returns K neighbors of $\mathbf{p}_i$, $\mathcal{A}$ aggregates the K neighbors, and $\mathcal{F}$ operates on the aggregation ($\mathbf{p}_i$ and its K neighbors) to generate the output $\mathbf{p}_o$.

The same formulation applies to the convolution operation in conventional CNNs as well, as illustrated in Fig. 6.2. However, the specifics of the three operations differ in point cloud networks and CNNs. Understanding the differences is key to identifying optimization opportunities.

Neighbor Search $\mathcal{N}$ in convolution returns K adjacent pixels in a regular 3D tensor by simply *indexing* the input feature map (K dictated by the convolution kernel



(a) Convolution in conventional CNNs can be thought of as two steps: 1) neighbor search ($\mathcal{N}$) by directly indexing adjacent pixels and 2) feature computation ($\mathcal{F}$) by a dot product.



(b) Point cloud networks consist of three main steps: neighbor search ($\mathcal{N}$), aggregation ($\mathcal{A}$), and feature computation ($\mathcal{F}$). $\mathcal{N}$ requires an explicit neighbor search; $\mathcal{A}$ normalizes neighbors to their centroid; $\mathcal{F}$ is an MLP with batched inputs (i.e., shared MLP weights).

Fig. 6.2: Comparing a convolution layer in conventional CNNs and a module in point cloud networks.

volume). In contrast, $\mathcal{N}$ in point cloud networks requires explicit *neighbor search* to return the K nearest neighbors of $\mathbf{p}_i$, because the points are irregularly scattered in the space. Similar to the notion of a "stride" in convolution, the neighbor search might be applied to only a subset of the input points, in which case N_{out} would be smaller than N_{in}, as is the case in Fig. 6.2b.

Aggregation Given the K pixels, convolution in CNNs directly operates on the raw pixel values. Thus, conventional convolution skips the aggregation step.

In contrast, point cloud modules operate on the *relative* value of each point in order to correlate a neighbor with its centroid. For instance, a point $\mathbf{p}_3$ could be a neighbor

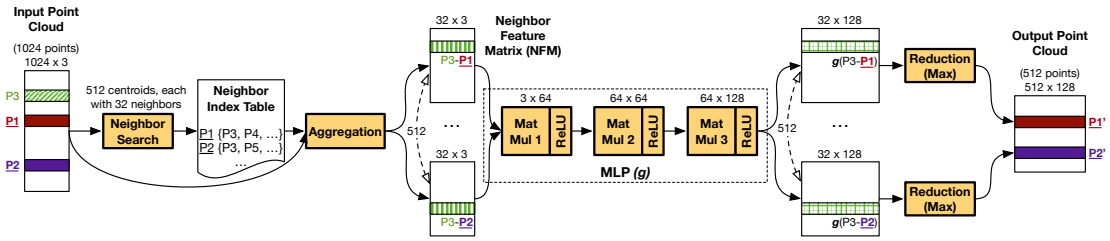


Fig. 6.3: The first module in PointNet++ (Qi et al., 2017b). The same MLP is shared across all the row vectors in a Neighbor Feature Matrix (NFM) and also across different NFMs. Thus, MLPs in point cloud networks process batched inputs, effectively performing matrix-matrix multiplications. The (shared) MLP weights are small in size, but the MLP activations are much larger. This is because the same input point is normalized to different values in different neighborhoods before entering the MLP. For instance, `P3` is normalized to different offsets with respect to `P1` and `P2` as `P3` is a neighbor of both `P1` and `P2`. In point cloud algorithms, most points are normalized to 20 to 100 centroids, proportionally increasing the MLP activation size.

of two centroids $\mathbf{p}_1$ and $\mathbf{p}_2$ (as is the case in Fig. 6.2b). To differentiate the different contributions of $\mathbf{p}_3$ to $\mathbf{p}_1$ and $\mathbf{p}_2$, $\mathbf{p}_3$ is *normalized* to the two centroids by calculating the offsets $\mathbf{p}_3 - \mathbf{p}_1$ and $\mathbf{p}_3 - \mathbf{p}_2$ for subsequent computations.

Generally, for each neighbor $\mathbf{p}_k \in \mathcal{N}(\mathbf{p}_i)$, the aggregation operation calculates the offset $\mathbf{p}_k - \mathbf{p}_i$ (a $1 \times M_{in}$ vector). All K neighbors' offsets form a Neighbor Feature Matrix (NFM) of size $K \times M_{in}$, effectively aggregating the neighbors of $\mathbf{p}_i$.

Feature Computation $\mathcal{F}$ in convolution is a dot product between the pixel values in a window and the kernel weights. In contrast, $\mathcal{F}$ in point cloud applies a multilayer perceptron (MLP) to each row vector in the NFM. Critically, all K row vectors share the same MLP; thus, the K input vectors are batched into a matrix and the MLP becomes a matrix-matrix product, transforming a $K \times M_{in}$ matrix to a $K \times M_{out}$ matrix.

In the end, a reduction operation then reduces the $K \times M_{out}$ matrix to a $1 \times M_{out}$ vector, which becomes the feature vector of an output point. A common choice for

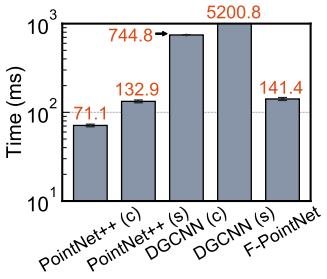


Fig. 6.4: Latency of five point cloud networks on the Pascal GPU on TX2. Results are averaged over 100 executions, and the error bars denote one standard deviation.

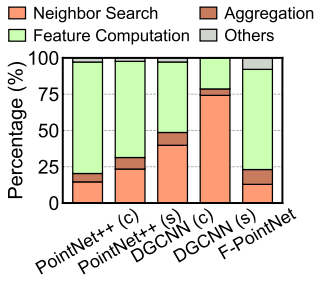


Fig. 6.5: Time distribution across the three main point cloud operations ($\mathcal{N}$, $\mathcal{A}$, and $\mathcal{F}$). The data is averaged on the mobile Pascal GPU on TX2 over 100 executions.

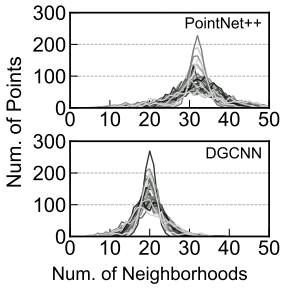


Fig. 6.6: Distribution of the number of points (y-axis) that occur in a certain number of neighborhoods (x-axis). We profile 32 inputs (curves).

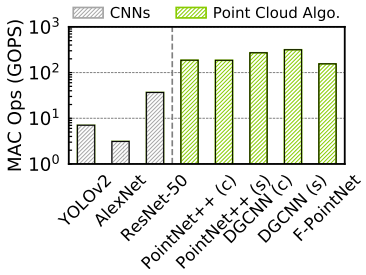


Fig. 6.7: MAC operation comparison between point cloud networks (130K input points per frame (Geiger et al., 2012a)) and conventional CNNs (nearly 130K pixels per frame).

reduction is to, for each column independently, take the max of the K rows.

Example Fig. 6.3 shows the first module in PointNet++ (Qi et al., 2017b), a classic point cloud network that many other networks build upon. This module transforms a point cloud with 1024 (N_{in}) points, each with a 3-D (M_{in}) feature vector, to a point cloud with 512 (N_{out}) points, each with a 128-D (M_{out}) feature vector, indicating that the neighbor search is applied to only 512 input points. Each neighbor search returns 32 (K) neighbors and forms a 32×3 NFM, which is processed by a MLP with 3 layers to generate a 32×128 matrix, which in turn is reduced to a 1×128 feature vector for

an output point. In this particular network, all the NFMs also share the same MLP.

Note that while feature computation is not always MLP and normalization is not always different from centroids, they are the most widely used, both in classic networks (e.g., PointNet++ (Qi et al., 2017b)) and recent ones (e.g., DGCNN (Wang et al., 2019a)).

6.2.2 Performance Characterizations

We characterize point cloud networks on today's systems to understand the bottlenecks and optimization opportunities. To that end, we profile the performance of five popular point cloud networks on the mobile Pascal GPU on the Jetson TX2 development board (Nvidia, 2017c), which is representative of state-of-the-art mobile computing platforms.

Time Distribution Fig. 6.4 shows the execution times of the five networks, which range from 71 ms to 5,200 ms, clearly infeasible for real-time deployment. The time would scale proportionally as the input size grows.

Fig. 6.5 further decomposes the execution time into the three components, i.e., Neighbor Search ($\mathcal{N}$), Aggregation ($\mathcal{A}$), and Feature Computation ($\mathcal{F}$). $\mathcal{N}$ and $\mathcal{F}$ are the major performance bottlenecks. While $\mathcal{F}$ consists of MLP operations that are well-optimized, $\mathcal{N}$ (and $\mathcal{A}$) is uniquely introduced in point cloud networks. Even if $\mathcal{F}$ could be further accelerated on a DNN accelerator, $\mathcal{N}$ has compute and data access patterns different from matrix multiplications (Xu et al., 2019), and thus does not fit on a DNN accelerator.

Critically, $\mathcal{N}$, $\mathcal{A}$, and $\mathcal{F}$ are serialized. Thus, they all contribute to the critical path latency; optimizing one alone would not lead to universal speedups. The serialization is inherent to today's point cloud algorithms: in order to extract local features of a point ($\mathcal{F}$), the point must be aggregated with its neighbors ($\mathcal{A}$), which in turn requires neighbor search ($\mathcal{N}$).

Memory Analysis Point cloud networks have large memory footprints. While the MLP weights are small and are shared across input NFMs (Fig. 6.3), the intermediate (inter-layer) activations in the MLP are excessive in size.

The "Original" category in Fig. 8.3 shows the distribution of each MLP layer's output size across the five networks. The data is shown as a violin plot, where the high and low ticks represent the largest and smallest layer output size, respectively, and the width of the violin represents the density at a particular size value (y-axis). The layer output usually exceeds 2 MB, and could be as large as 32 MB, much greater than a typical on-chip memory size in today's mobile GPUs or DNN accelerators. The large activation sizes would lead to frequent DRAM accesses and high energy consumption.

The large activation size is fundamental to point cloud algorithms. This is because an input point usually belongs to many overlapped neighborhoods, and thus must be normalized to different values, one for each neighborhood. Fig. 6.2b shows a concrete example, where `P3` is a neighbor of both `P1` and `P2`; the aggregation operation normalizes `P3` to `P1` and `P2`, leading to two different relative values (`P3 - P1` and `P3 - P2`) that participate in feature computation, increasing the activation size. This is in contrast to convolutions, where pixels in overlapped neighborhoods (windows) are directly reused in feature computation (e.g., `P4` in Fig. 6.2a).

We use two networks, DGCNN (Qi et al., 2017b) and PointNet++ (Wang et al., 2019a), to explain the large activation sizes. Fig. 6.6 shows the distribution of the number of neighborhoods each point is in. Each curve corresponds to an input point cloud, and each (x, y) point on a curve denotes the number of points (y) that occur in a certain number of neighborhoods (x). In PointNet++, over half occur in more than 30 neighborhoods; in DGCNN, over half occurs in 20 neighborhoods. Since the same point is normalized to different values in different neighborhoods, this bloats the MLP's intermediate activations.

Compute Cost The large activations lead to high multiply-and-accumulate (MACs) operations. Fig. 6.7 compares the number of MAC operations in three classic

CNNs with that in the feature computation of point cloud networks. To use the same "resolution" for a fair comparison, the input point cloud has 130,000 points (e.g., from the widely-used KITTI Odometry dataset (Geiger et al., 2012a)) and the CNN input has a similar amount of pixels. In feature computation alone, point cloud networks have an order of magnitude higher MAC counts than conventional CNNs.

Summary Today's point cloud algorithms extract local features of a point by aggregating the point with its neighbors. The aggregation happens *before* feature computation, which leads to **two fundamental inefficiencies in compute**:

- The two major performance bottlenecks, neighbor search and feature computation, are serialized.
- Feature computation operates on aggregated neighbor points, leading to high memory and compute costs.

6.3 Memory Inefficiencies in Point Cloud Analytics

This section quantifies the memory inefficiencies in two main operations of point cloud analytics, neighbor search (Sec. 6.3.1) and feature computation (Sec. 6.3.2).

6.3.1 Memory Inefficiencies in Neighbor Search

Neighbor search in low-dimensional space (e.g., 3D) commonly uses the K-d tree (Bentley, 1975), which recursively subdivides the search space into two half-spaces using axis-aligned planes. The sub-spaces are organized as a tree, and the neighbor search becomes a tree traversal problem. Compared to exhaustive search, the space subdivision strategy is more efficient as it prunes the search space: if the distance of a query $\mathcal{Q}$ and the boundary of a subspace $\mathbb{S}$ is greater than the search radius, all the points in $\mathbb{S}$ can be skipped.

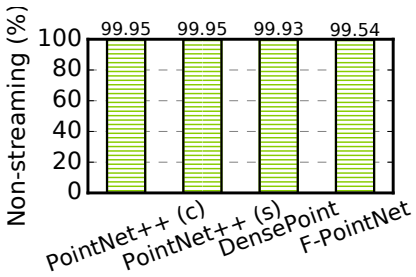


Fig. 6.8: Percentage of non-continuous DRAM accesses in common point cloud networks.

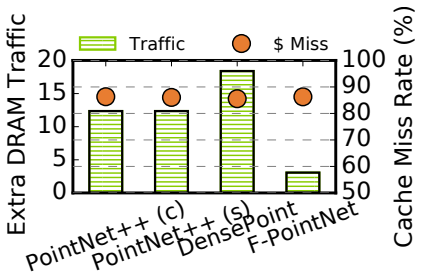


Fig. 6.9: Ratio of actual DRAM traffic vs. the theoretical minimum and cache miss rate in neighbor search.

While K-d tree search is inherently parallel (as different search queries are independent), tree traversals are hardware unfriendly. In particular, the memory access patterns are known only at run time, leading to massive inefficiencies in both DRAM and SRAM accesses, which we quantify below.

DRAM DRAM access inefficiency in neighbor search is manifested in two ways: non-streaming accesses and redundant accesses. The DRAM accesses are non-streaming because the inputs (points) are arbitrarily distributed in the search space. If two queries being processed in parallel are spatially far-apart, they will likely exercise different parts of the K-d tree that are non-contiguous in memory. Even within the same query, tree nodes consecutively accessed during traversals are likely non-continuous in memory due to the control-flow heavy nature of tree traversal. Fig. 6.8 shows the percentage of non-continuous DRAM accesses across four popular point cloud DNNs. Almost all DRAM accesses are non-continuous.

The non-streaming nature coupled with large point cloud data size leads to redundant DRAM accesses. For instance, in the popular KITTI dataset (Geiger et al., 2012a), the total points and queries in a typical scene *alone* can be over tens of MBs (not considering the network weights, activations, etc.), larger than what a mobile SoC can accommodate. Thus, points are loaded on-chip in chunks (analogous to tiling in conventional DNNs). Since not all data in each chunk will be used when they are loaded

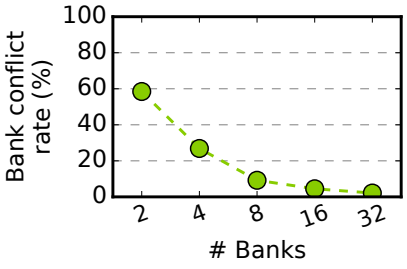


Fig. 6.10: Neighbor search bank conflict rate in Pointnet++(c) vs. the number of banks under 8 concurrent queries.

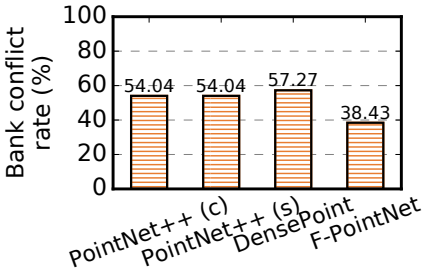


Fig. 6.11: SRAM bank conflict rate in aggregation, assuming 16 banks and 16 concurrent memory requests.

due to the non-streaming access pattern, a great amount of DRAM accesses are wasted.

Fig. 6.9 quantifies the excessive DRAM accesses and cache miss rate in neighbor search. The left y-axis shows the ratio of the amount of DRAM requests (in bytes) to the actual data theoretically needed by the algorithm (i.e., reading each query and search point once). The data are obtained by simulating an unrealistic 10 MB fully-associated cache running a neighbor search on a typical KITTI-constructed scene with about 1.2 million points. Even with this unrealistic SRAM structure, searches in many models have about 10$\times$ more DRAM traffic than what is strictly required. Realistic mobile accelerators would allocate an even smaller buffer for neighbor search to accommodate other data structures such as DNN weights and activations. The right y-axis quantifies the corresponding cache miss rates, which are over 85%.

SRAM The on-chip memory accesses in K-d tree search are also inefficient because of the frequent bank conflicts. In regular kernels such as stencil pipelines (Qadeer et al., 2013; Whatmough et al., 2019a) where the memory access pattern is statically known, one could carefully interleave data in the SRAM banks to avoid bank conflicts (Kirk and Wen-Mei, 2016; Zhou et al., 2021). In contrast, on-chip memory accesses in neighbor search are input-dependent, thus, bank conflicts are inevitable.

Fig. 6.10 quantifies the bank conflicts by showing the percentage of SRAM accesses that are bank-conflicted and how the percentage varies with the number of banks. We

assume an unrealistically large 10 MB buffer and 8 concurrent SRAM requests. With 4 banks the bank conflict rate is 26.9%. The bank conflict rate is reduced to 2.1% only when the number of banks quadruples the number of simultaneous requests.

Using a heavily-banked SRAM design is highly undesirable. A large number of banks requires a more costly crossbar design (Agarwal et al., 2009; Grot et al., 2011), as the crossbar area grows quadratically with the number of banks. Using an Arm memory compiler (Arm, 2016), we find that the crossbar area is twice as much as the memory arrays under a 32-bank configuration. In addition, a higher bank count also reduces the memory array size, which increases the per-bank overhead (peripheral circuits, BIST, redundancy) (Weste and Harris, 2015).

6.3.2 Memory Inefficiencies in Feature Computation

Unlike neighbor search, the DRAM accesses in the feature computation stage are completely streaming. The on-chip memory accesses, however, are met with frequent bank conflicts.

Feature computation is broken down into two steps: 1) aggregate the neighbors for each input point $\mathbf{p}_i$ using the neighbor indices generated in the neighbor search stage, and 2) compute an output point $\mathbf{p}_o$ from each $\mathbf{p}_i$ by applying a function, usually a MLP, to the neighbors of $\mathbf{p}_i$. Step 2 is accelerated on today's DNN accelerators.

Step 1 is analogous to fetching data from the input feature map in a conventional DNN. However, conventional DNNs access consecutive feature map elements with statically-known patterns. Therefore, a compiler lays out data in the SRAM such that a simple single-bank, single-port memory array (using wide words) could serve memory requests from tens or hundreds of PEs in one cycle without stalling the PEs (Jouppi et al., 2017b; Arm, 2018; Zhou et al., 2021).

However, point cloud networks access non-consecutive memory in this step, because the neighbors of a point can be arbitrary. Therefore, the SRAM serving points

are usually banked. Worse, the access pattern is statically-unknown, as it depends on the neighbor search results, which, in turn, depend on the inputs. Therefore, bank conflicts are inevitable.

Fig. 6.11 quantifies the severity of bank conflicts in point aggregation by showing the percentage of SRAM accesses that are bank-conflicted in aggregating the points. We assume a 16-bank SRAM design with a total size of 64 KB. Across the four models, the bank conflict rate is at least 38.43% and can be as high as 57.27%. Increasing the number of banks is undesirable as it requires a more costly crossbar and/or a higher per-bank overhead due to the smaller memory arrays (Weste and Harris, 2015).

6.4 Summary

Despite point cloud being extensively used in many mobile applications, there are still many issues that prevent point cloud from being fully adopted in many potential use cases. Here, I summarize several remaining challenges in point cloud applications:

- **Data Communication**: Unlike images, point clouds do not have well-established compression schemes from software interfaces to hardware supports. While various compression schemes have been proposed (Huang et al., 2006; Kammerl et al., 2012; Hornung et al., 2013; Thanou et al., 2016; Lasserre et al., 2019; Jang et al., 2019; Krivokuća et al., 2020), no single compression standard has been widely adopted as the industry standard. This lack of consensus poses a significant challenge in designing hardware to achieve better energy efficiency.

- **Hardware Support for Computation**: Although point-based DNN computation can achieve great accuracy with lower memory utilization (Liu et al., 2019c; Feng et al., 2020b), direct point manipulation also poses a great challenge to today's hardware accelerators. How to compensate for neighbor search remains an active research question for the architecture community.

- **Irregular Memory Access**: Although key operations in point-based DNN computation are computationally effective, their memory accesses are irregular. This causes memory inefficiencies in both SRAM and DRAM accesses, thus, posing a challenge for hardware acceleration.

The following chapters propose different techniques to address the aforementioned inefficiencies.

7 Reducing Data Communication via Spatio-Temporal Compression

As mentioned in Sec. 6.1, massive amounts of point cloud data are generated for many real-time LiDAR-involved vision applications. To alleviate this data communication pressure, this chapter introduces a real-time point cloud compression algorithm. The main idea of this compression method is explained in Sec. 7.1. Sec. 7.2 and Sec. 7.3 quantitatively evaluates the effectiveness of this method compared against state-of-the-art methods. Lastly, Sec. 7.4 highlights related work in this area.

7.1 Spatio-Temporal Compression

This section introduces our spatial-temporal LiDAR point cloud compression algorithm. We first present an overview of our compression system (Sec. 7.1.1), followed by the detailed designs of the three key components: range image conversion (Sec. 7.1.2), spatial encoding (Sec. 7.1.3), and temporal encoding (Sec. 7.1.4). Finally, we discuss our parallel implementation that further improves the compression speed (Sec. 7.1.5).

7.1.1 Main Idea

The idea of our compression system is to exploit redundancies both within a point cloud (spatial) and across point clouds (temporal). Spatially, many surfaces in the real-world

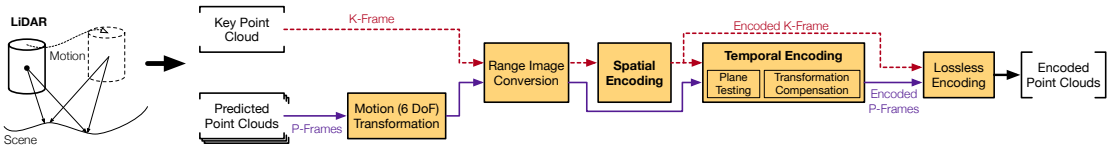


Fig. 7.1: Overview of our compression system, which compresses a sequence of consecutive point clouds. All the points clouds are converted to range images to accelerate the compression speed. We first spatially encode the key point cloud (K-frame) in the sequence, typically the middle one. The spatial encoding results of the K-frame are then used to temporally encode the rest of the point clouds, which we call predicted point clouds (P-frames).

are planes (e.g., walls and ground); even non-plane surfaces could be approximated by a set of planes. Temporally, consecutive point clouds share a great chunk of overlapped areas of the scene; thus, the same set of planes could be used to encode points across point clouds. While intuitive, exploiting spatial and temporal redundancies in real-time is challenging due to the irregular/unstructured point cloud and the compute-intensive plane fitting process.

We propose a compression system that simultaneously achieves the state-of-the-art compression rate and compression speed while maintaining high application accuracies. Fig. 7.1 provides a high-level overview of our system, which consists of three main blocks: range image conversion, spatial encoding, and temporal encoding. Fig. 7.2 shows the relevant data structures during the encoding process.

Given a sequence of consecutive point clouds, we differentiate between two point cloud types: key point cloud (K-frame) and predicted point cloud (P-frame). A sequence has only one K-frame and the rest is P-frames. P-frames are first transformed (both translation and rotation) to K-frame's coordinate system using the IMU measurements. After transformation, each point cloud is converted to a range image (Tu et al., 2016) for subsequent computations. The range image not only provides an initial compression to the original point cloud, but also provides a structured representation of the

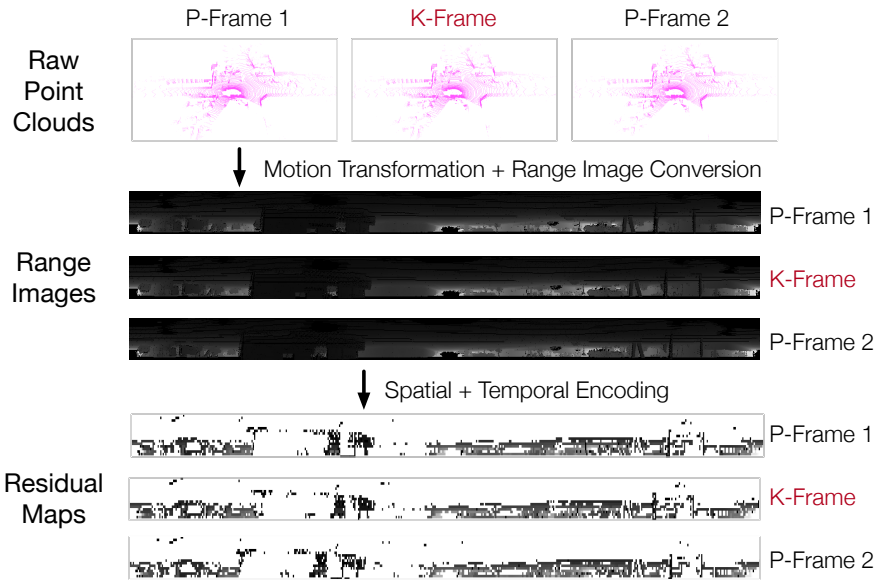


Fig. 7.2: Different data structures used in our compression. The raw point clouds are converted to range images. After spatial and temporal encoding, most of the tiles in the range images are plane-encoded; the unfit tiles are left in the residual maps.

(unstructured) point cloud that is hardware-friendly.

We then spatially encode K-frame by fitting planes; the fitted planes in the K-frame are then (re-)used to temporally encode P-frames, greatly improving the overall compression rate and speed. In order to be robust against transformation errors, which might be introduced due to noisy IMU observations, we propose a set of techniques that compensate the sensor noise and preserve the encoding quality.

In the end, after spatial and temporal encoding, most of the tiles in the range images are plane-encoded; the unfit tiles are left in what we call residual maps (Fig. 7.2). The planes and the residual maps are then further compressed by a lossless compression scheme (e.g., Huffman encoding) to generate the final encoded data.

Overall, in addition to providing high compression rate and speed, our compression system also preserves application accuracy. This is because plane fitting inherently removes noises and outliers in the point clouds without requiring explicitly removing

outliers that prior work employs (Sun et al., 2019).

7.1.2 Range Image Conversion

We first convert the raw point cloud data to a range image, which essentially converts every point (x, y, z) in the 3D Cartesian space to a pixel at coordinates (θ, ϕ) in the range image with a pixel value r:

$$r = \sqrt{x^2 + y^2 + z^2}; \tag{7.1}$$

$$\theta = arctan(\frac{x}{y})/\theta_r;\ \phi = arccos(\frac{z}{r})/\phi_r \tag{7.2}$$

where θ_r and ϕ_r are the horizontal and vertical resolutions of the LiDAR, respectively.

A range image naturally compresses the original point cloud, because each point (x, y, z) can be encoded with just a range value r of the corresponding pixel in the range image; θ and ϕ are the pixel's coordinates and do not have to be explicitly encoded. If θ_r and ϕ_r are the same as the resolutions of the LiDAR, range image is a lossless compression of the corresponding point cloud. Mathematically, however, θ_r and ϕ_r could be any arbitrary positive values; larger θ_r and ϕ_r would lead to a lower range image resolution, providing a lossy compression of the original point cloud.

In addition to providing an inherent compression scheme, range image brings two key advantages. First, operating on range images is computationally more efficient than directly accessing the point cloud, which requires tree traversals that lead to high cache misses and branch mis-predictions on today's hardware architecture (Xu et al., 2019; Liu et al., 2019c). Second, adjacent pixels in the range map are likely to lie on the same plane, because they correspond to consecutive scans from the LiDAR. This characteristic allows us to encode the entire range image more efficiently.

7.1.3 Spatial Encoding

The goal of spatial encoding is to encode all the points that lie on the same plane using that plane. Intuitively, many surfaces in the real world are planes (e.g., walls and ground); non-plane surfaces could be approximated by a set of planes.

In the 3D Cartesian space, a plane can be expressed as:

$$x + ay + bz - c = 0 \tag{7.3}$$

where $(1, a, b)$ is the normal vector of the plane and $\frac{|c|}{\sqrt{1+a^2+b^2}}$ is the distance from the origin (LiDAR center) to the plane. Thus, all the points on the same plane could be encoded with just the three coefficients of the plane. Note that the exact position of each point on the plane is not explicitly encoded. The decoding process would simply have to simulate a ray casting process to find the intersection of a ray and the plane to reconstruct the position of a point.

To encode the entire point cloud, which contains points that lie in many different planes, we use a "divide and conquer" strategy. Specifically, we first uniformly divide the range image into unit tiles (e.g. 4×4). We start by fitting a plane for points in the first tile, and gradually grow to include adjacent tiles, essentially forming a bigger tile. Each time we grow, we test whether the plane fit so far can be used to encode all the points in the new (bigger) tile under a predefined threshold. If so, all the points in the new tile are encoded with the plane. Otherwise, we start from the current tile and repeat the process until all the tiles in the range image are processed.

Our spatial encoding process grows tiles horizontally, which we find coalesces many more adjacent tiles than growing vertically. This is inherently because today's LiDARs have a much more fine-grained horizontal resolution than vertical resolution. For instance, Velydone's HDL-64E has a 0.08° horizontal resolution, and a 0.4° vertical resolution. As a result, points in horizontally adjacent tiles are closer to each other and, thus, more likely to fit in the same plane than points from vertically adjacent tiles.

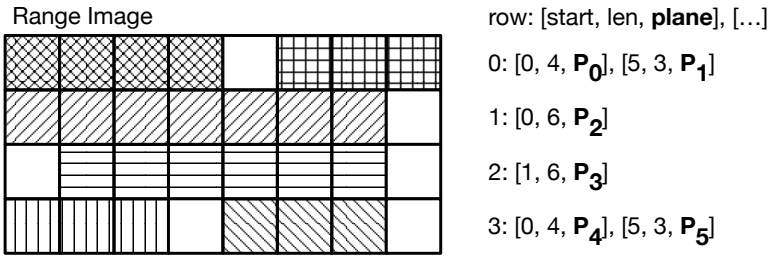


Fig. 7.3: A spatial encoding example. The range image on the left is first tiled, and then iteratively planed-fitted. Horizontally adjacent tiles fit by the same plane are shaded by the same stripe pattern. Tiles that are plane-fitted are encoded using the format shown on the right. Points in unfit tiles are encoded individually using their range values (not shown).

Fitting a plane given points can be naturally formulated as a linear least squares problem (Nievergelt, 1994). While classic iterative methods such as RANSAC (Fischler and Bolles, 1981) are widely used, we find that directly calculating the closed-form solution is generally faster, because deriving the closed-form solution requires less computation and also the computations could be parallelized.

Note that we intentionally do not encode the deltas of plane fitting (i.e., the difference between a true point and a predicted point on the plane). Instead, we find that when a reasonably small threshold is used, discarding deltas effectively *denoises the point cloud*, leading to higher application accuracy than even the original point cloud (Sec. 7.3.1).

In order to reconstruct/decode the range image later, each row in the range image is encoded with a row ID followed by a set of three-tuples $[s, len, \mathbf{P}]$. Fig. 7.3 provides an example. Each three-tuple corresponds to a sequence of adjacent tiles in that row, starting from s to $s + len$, that are fit by the same plane $\mathbf{P}$, which is parameterized by the three coefficients (Equ. 7.3). Inevitably, there are tiles that contain points that can not be fit on planes, because, for instance, those points are sparse samples of an irregular surface. These "unfit" points are left in what we call a residual map (Fig. 7.2)

and are directly encoded using their raw range values.

7.1.4 Temporal Encoding

Spatial encoding provides a building block to encode point clouds individually. LiDARs in autonomous machines, however, generate a sequence of point clouds. While it is possible to individually apply spatial encoding to each point cloud, doing so loses opportunities exposed by the temporal correlations across consecutive point clouds.

Consecutive point clouds have large chunks of overlaps, because they are just different samples of the same physical scene. Using the KITTI dataset, we find that on average 99% of each point cloud is geometrically overlapped with the previous point cloud. Motivated by this observation, temporal encoding encodes a set of consecutive point clouds together. The idea is to use one plane to encode the overlapped scene across multiple consecutive point clouds. Doing so improves both the compression rate and the compression speed by avoiding plane fitting in each point cloud.

Transformation Each point cloud has its own coordinate system when generated by the LiDAR. In order to fit planes across a sequence of point clouds, we convert all the point clouds to the same coordinate system—by performing a 6 DoF (translation and rotation) transformation. Fig. 7.4 compares the effect of overlaying five consecutive point clouds together in the same coordinate system before and after the transformation. Without motion transformation point clouds at different timestamps are misaligned, making temporal encoding challenging.

To unify point clouds in the same coordinate system, we must 1) decide a key point cloud K, whose coordinate system is used as the transformation target, and 2) calculate the corresponding transformation matrix $\mathbf{M}_i$ between K and every other point cloud P_i.

In our system, we calculate the transformation matrix using the IMU measurements, which provides the translational acceleration ($\hat{\mathbf{a}}$) and rotational rate ($\hat{\omega}$). Using the IMU

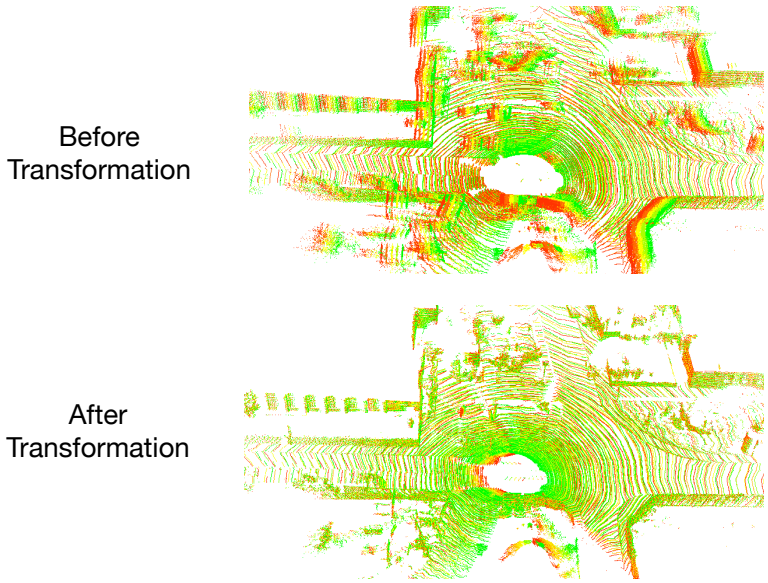


Fig. 7.4: Stacking five consecutive point clouds before (top) and after (bottom) motion transformation. Colors indicate different point clouds. Motion transformation better aligns different point clouds in one coordinate system.

measurements, we estimate the translation vector $T_{3\times1}$ as:

$$T_{3\times1} = \left[\overline{\Delta x} \quad \overline{\Delta y} \quad \overline{\Delta z}\right] \tag{7.4}$$

where $\overline{\Delta x}$, $\overline{\Delta y}$, and $\overline{\Delta z}$ are translational displacements integrated from $\hat{\mathbf{a}}$ using the first-order Runge-Kutta numerical method. Similarly, the rotation matrix $R_{3\times3}$ is estimated

as:

$$
\begin{aligned}
R_{3\times3} = & \begin{bmatrix} cos(\overline{\Delta\alpha}) & sin(\overline{\Delta\alpha}) & 0 \\ -sin(\overline{\Delta\alpha}) & cos(\overline{\Delta\alpha}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
& \times \begin{bmatrix} cos(\overline{\Delta\beta}) & 0 & -sin(\overline{\Delta\beta}) \\ 0 & 1 & 0 \\ sin(\overline{\Delta\beta}) & 0 & cos(\overline{\Delta\beta}) \end{bmatrix} \\
& \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\overline{\Delta\gamma}) & sin(\overline{\Delta\gamma}) \\ 0 & -sin(\overline{\Delta\gamma}) & cos(\overline{\Delta\gamma}) \end{bmatrix}
\end{aligned} \tag{7.5}
$$

where $\overline{\Delta\alpha}$, $\overline{\Delta\beta}$, and $\overline{\Delta\gamma}$ are rotational displacements integrated from $\hat{\omega}$ using the first-order Runge-Kutta method.

We use the middle point cloud in a consecutive point cloud sequence as the key point cloud (K-frame). This minimizes the impact of cumulative IMU sample errors when calculating the transformation matrix. Every other point cloud, which we call predicted cloud (P-frame), is transformed to K-frame's coordinate system by:

$$
p'_{4\times1} = \mathbf{M} p_{4\times1} = \begin{bmatrix} R_{3\times3} & T_{3\times1} \\ 0_{1\times3} & 1 \end{bmatrix}_{4\times4} p_{4\times1} \tag{7.6}
$$

where $p_{4\times1}$ and $p'_{4\times1}$ denote a point in a predicted cloud before and after transformation, respectively.

All the N point clouds in a sequence, after transformation, are converted to range images with the same dimension. For ease of manipulation, we stack the N range images together to form a N-channel image.

Note that it is possible that points in a P-frame after transformation could collide, i.e., mapped to the same range image pixel, in which case we preserve the nearest point.

On the KITTI dataset, about 4.6% of the points collide when transforming between two adjacent point clouds, and this percentage increases as the gap between two point clouds increases. This suggests that the number of consecutive point clouds that are encoded together (N) affects the encoding results. We show the sensitivity to N in Sec. 7.3.3.

Encoding We use the same "divide-and-conquer" strategy used in spatial encoding to encode across channels (point clouds). A naive implementation would be to fit all the points in a tile across all N channels (e.g., $4 \times 4 \times N$) and then grow to adjacent tiles. However, this approach is susceptible to IMU measurement errors. Inaccurate IMU observations lead to inaccurate point cloud transformations. As a result, points in the same tile across different channels might not end up lying on the same plane, leading to poor plane fitting results.

We propose an effective method to temporally encode across channels while compensating for the transformation errors. Our idea is to first fit a tile in the K-frame's channel, and use the fitted plane Q to test against the same tile in each of the other channels. Critically, we hold Q's normal vector constant while varying its distance to the origin (i.e., varying c in Equ. 7.3). This effectively compensates for the translation error in the IMU measurements. If the relaxed plane Q' (parameterized by a, b, c') fits all the points in a channel under a certain threshold, only c' needs to be encoded for that channel rather than all three plane coefficients.

We apply the same horizontal growing strategy until all the tiles of all the channels in the range image are processed, at which point we remove all the encoded tiles from the range image. The remaining range image I' contains tiles that could not be fitted across channels even after compensating translation errors. We then spatially encode I' channel by channel using the same process described in Sec. 7.1.3. Effectively, this channel-wise spatial encoding compensates for the rotation errors in transformation. In the end, the unfit tiles are left in the residual map (Fig. 7.2), which is further compressed in a lossless fashion along with the fit planes.

Not only does Temporal encoding provide a high compression rate, but it also im-

proves the compression speed compared to spatially compressing each point cloud individually. This is because the planes that fit in the K-frame are reused in P-frames, reducing the plane fitting overhead.

7.1.5 Parallel Optimizations

The speed of the sequential implementation of our algorithm scales linearly with respect to the number of channels and angular resolution of the LiDAR. To further improve the compression speed, we exploit the parallelisms exposed by our encoding system and leverage parallel hardware available in modern processors.

At the high level, we exploit both thread-level parallelism (TLP) and data-level parallelism (DLP). During the range image conversion, we exploit the TLP where each thread is responsible for converting one point cloud into the corresponding range image. During spatial encoding, we leverage TLP where each thread is responsible for encoding a row in the K-frame. During temporal encoding, each thread is responsible for testing planes in a P-frame.

The actual computation in each thread also exposes data-level parallelism such as computing the immediate results (radius, indexes) and the various matrix operations in the plane-fitting and plane-testing processes. Our implementation uses the OpenMP programming model in C++ to exploit both TLP and DLP.

7.2 Evaluation Methodology

Applications and Evaluation Metrics We evaluate our compression method on three common point cloud applications: registration, object detection, and scene segmentation:

- Registration: we use a recent ICP-based registration pipeline (Xu et al., 2019) developed using the widely-used PCL (Rusu and Cousins, 2011).

- Object Detection: we use VoxelNet (Zhou and Tuzel, 2018), a Deep Convolution Neural (DNN)-based approach.
- Scene Segmentation: we use SqueezeSeg (Wu et al., 2018a), a DNN-based approach.

We use three evaluation metrics: compression rate over the uncompressed point clouds, compression speed in FPS, and application-level accuracy. We evaluate the application-level accuracy instead of common quality metrics such as PSNR or RMSE because we want to assess how compression affects point cloud applications, which is what ultimately matters.

Dataset We use the widely-used KITTI dataset (Geiger et al., 2012b) for evaluating registration and object detection. We evaluate on all the sequences and frames for comprehensiveness. To evaluate segmentation, we use SemanticKITTI (Behley et al., 2019), which augments KITTI dataset for segmentation tasks. We report geometric mean results unless otherwise noted.

Baseline We compare against four baselines:

- G-PCC: It is a point cloud compression standard proposed by the MPEG (Lasserre et al., 2019) specifically designed to compress LiDAR point cloud data. It constructs an Octree for a point cloud and encodes the Octree.
- V-PCC: It is a point cloud compression standard proposed by the MPEG (Jang et al., 2019; Krivokuća et al., 2020) designed to compress dense point clouds used in volumetric rendering. It maps point clouds to images and uses existing video compression to compress the images.
- JPEG: It compresses each point cloud's range image individually using the (lossy) JPEG codec (ITU, 2020).

- H.264: It compresses a sequence of point clouds by compressing the corresponding range image sequence using the H.264 video codec (Wiegand et al., 2003). We shows results of both the lossy and lossless versions.

Variants Our method can be configured in two modes: the *single-frame* mode that applies only spatial encoding to individual frames and the *streaming* mode that applies both spatial and temporal encoding to a sequence of frames. For both versions, we vary the threshold of plane fitting to form different design points.

Hardware Platform We implement our compression method in C++ and evaluate the compression speed on both a PC, Intel i5-7500 with 4 cores, and a mobile platform, Nvidia Jetson TX2 (Nvidia, 2017c), which represents the compute capability of mobile robots or drones.

7.3 Evaluation

We first show the end-to-end accuracy and compression rate of our compression method on three general robotic applications: localization, object detection, and 3D scene segmentation, compared against a range of existing methods (Sec. 7.3.1). We then demonstrate that our compression speed matches the point cloud generation speed and surpasses other methods (Sec. 7.3.2). Last, we evaluate the sensitivity of our compression method (Sec. 7.3.3).

7.3.1 Compression Rate vs. Accuracy

This section assumes that we compress five consecutive point clouds together unless otherwise noted. We will later study the sensitivity of different frame configurations.

Localization Our compression method outperforms other methods in both application accuracy and compression rate. Fig. 7.5 compares the translation error (y-axis) against the compression rate (x-axis) of different compression methods.

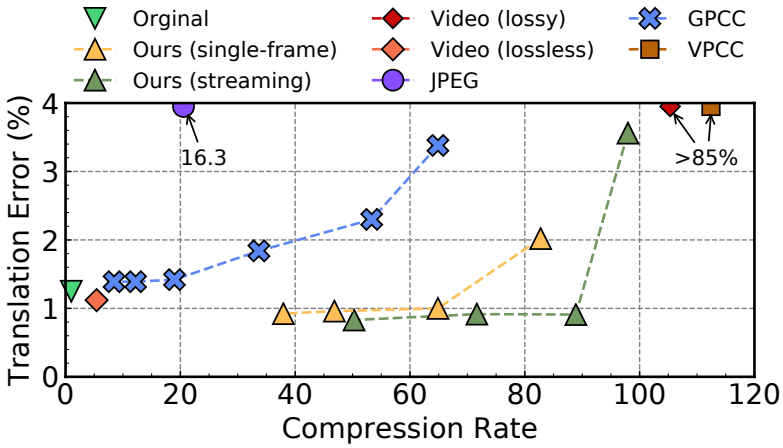


Fig. 7.5: Registration translation error and compression rate comparison of various compression methods.

Our method in the streaming mode can achieve an 88.9× compression rate with only 0.91% translation error, and the single-frame mode achieves 59.7× compression rate with 0.96% translation error. In comparison, the best G-PCC compression has a 1.38% translation error with only 8.5× compression rate. Interestingly, our compression methods have lower errors than using the original point clouds (1.25%). This is because our plane fitting process inherently reduces the noise from the point cloud.

Other baselines including JPEG compression on range images, lossy H.264 video compression, and V-PCC have much higher localization errors ($>$ 16%) as Fig. 7.5 shows. Although the lossless video compression has better localization accuracy, its compression rate (5.4×) is much lower.

Object Detection On KITTI dataset object detection uses only individual point clouds instead of point cloud sequences. Thus, we present only the single-frame variant of our compression system. For the same reason, V-PCC and H.264 compression methods are not applicable. Fig. 7.6 compares the object detection accuracy against compression rate across different compression methods. Our method Pareto-dominates the prior methods.

Comparing against the 74.4% accuracy using the original point clouds, our com-

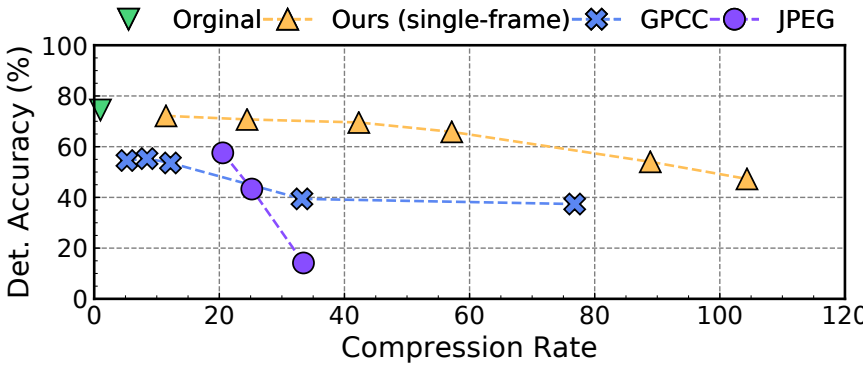


Fig. 7.6: The object detection accuracy and compression rate comparison of various compression methods.

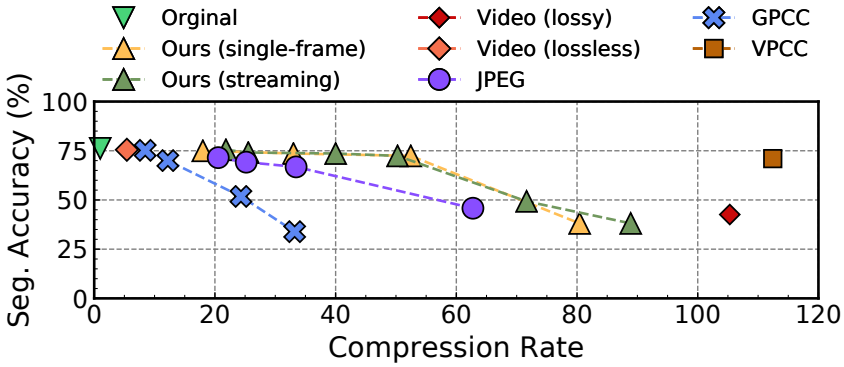


Fig. 7.7: The segmentation error and compression rate comparison of various compression methods.

pression method achieves a comparable accuracy of 72.2% with an 11.5× compression rate. In addition, our compression method achieves more than 42.3× compression rate while still keeping the accuracy over 70%. In contrast, the best accuracy that G-PCC and JPEG achieve is 42.1% and 57.6% with compression rates of 15.3 and 20.6, respectively.

Segmentation Fig. 7.7 shows the compression rate vs. segmentation accuracy trade-offs across the different compression schemes. Our method Pareto-dominates other methods except for lossless video compression. In particular, our method achieves a better compression rate (21.8×) than G-PCC (8.5×) and JPEG (20.6×) with a similar

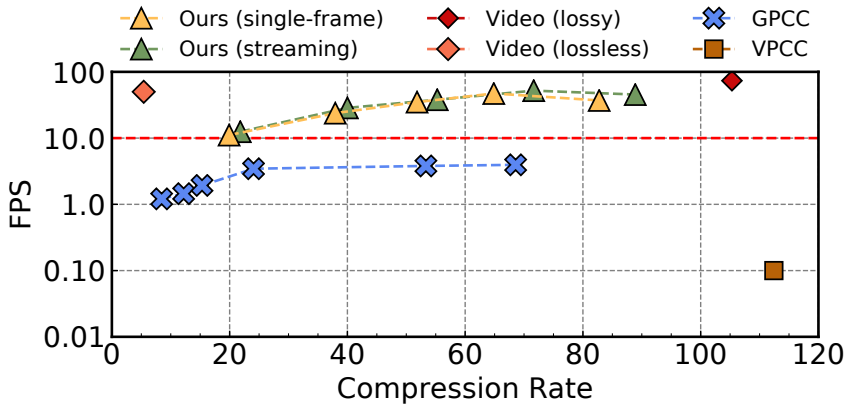


Fig. 7.8: Compression speed vs. compression rate of various methods on Intel i5-7500 CPU.

accuracy at 75.5%. The accuracies of G-PCC and JPEG drop quickly as the compression rates increase while our method maintains a high accuracy (72.4%) even at a compression rate of 50.3×.

Lossless video compression achieves little accuracy drop with only a 5.4× compression rate; lossy video compression, in contrast, has the highest compression rate—at the expense of over 30% accuracy drop.

7.3.2 Compression Speed

Fig. 7.8 and Fig. 7.9 show the compression speeds on both a PC and the Nvidia TX2 mobile platform, respectively. Our compression method outperforms G-PCC by about one order of magnitude on both platforms. The compression speed on a PC could be as high as 52.1 FPS, and even on the mobile TX2 the compression speed could be as high as 20.5 FPS. As today's LiDARs generally operate at between 5 Hz to 20 Hz (Velodyne, 2017, 2020), our compression method could be executed in real-time as the point clouds are being generated. Lossy and lossless video compressions have a similar compression speed. However, as shown before, they either have a much lower compression rate or

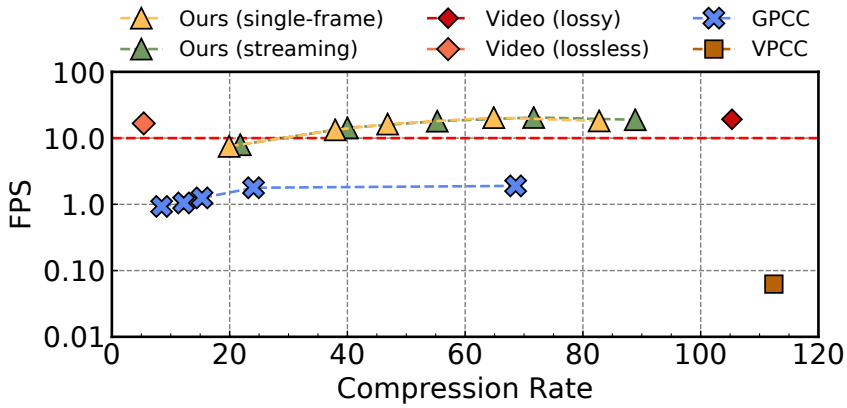


Fig. 7.9: Compression speed vs. compression rate of various methods on Nvidia mobile TX2 platform.

lead to much lower application accuracies. V-PCC is much slower than other methods.

7.3.3 Sensitivity Study

All results shown so far assume that five consecutive point clouds are encoded together. Fig. 7.10 shows how compression rate, application accuracy, and compression rate vary with the number of consecutively encoded frames. All results are normalized to the results where the number of consecutive frames is five. Object detection uses individual frames, so its accuracy numbers are not shown.

We find that the compression speed is most sensitive to the number of encoded frames. This is because our implementation parallelizes many operations across frames such as the range image conversion and plane testing. More frames provide more opportunities for parallelization, leading to higher speeds. The application accuracies are mostly insensitive to the number of frames, because our compression method is able to preserve the vast majority of points during motion transformation and encoding.

It is worth noting that the compression rate is relatively insensitive to the number of frames. To understand why, Fig. 7.11 shows the distribution of how different points are

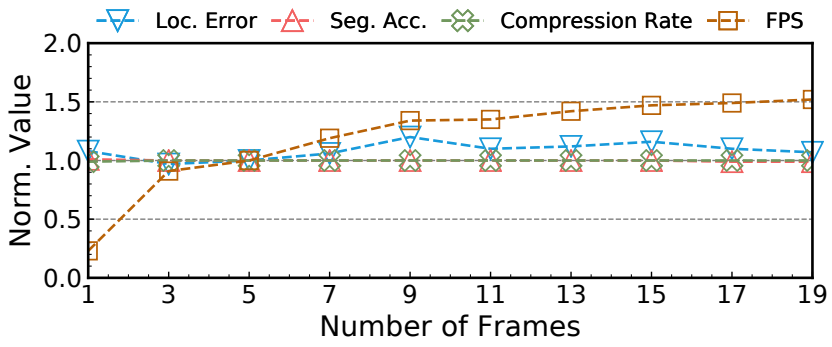


Fig. 7.10: Sensitivity study on application accuracy, compression rate, and compression speed by varying the number of consecutive frames that are encoded together.

encoded in a sequence. As the number of frames increases, the percentage of temporally encoded points decreases because the overlapped region becomes smaller, while the percentage of spatially encoded points increases. The overall percentage of points that are encoded by either method stays roughly the same, leading to a roughly stable compression rate. Note that as the number of frames increases the decoding speed is faster with similar accuracy as shown in Fig. 7.10, indicating longer sequences are preferred in encoding point clouds.

Fig. 7.12 shows the speedup of our parallel compression system over a sequential baseline. Recall from Sec. 7.1.5 that our implementation exploits various forms of parallelism to improve the speed. With five frames available for compression, we achieve a 3.8× speedup over a sequential implementation. With 19 frames available, the speedup is 5.4×. As the number of consecutive frames increases, the speedup saturates because of the hardware resource limitation.

7.4 Related Work

Unstructured Point Cloud Encoding Perhaps the most common way to encode point cloud data is to use space-partitioning trees, among which Octree is the most widely

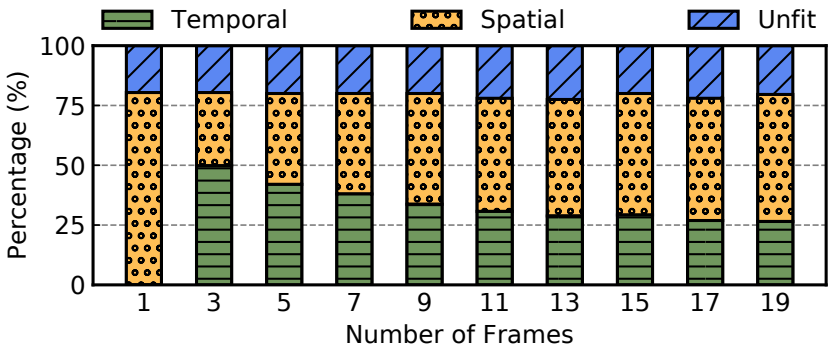


Fig. 7.11: Distribution of different encoding types within a point cloud sequence as the number of consecutively encoded frames varies. "Unfit" refers to points that could not be encoded in either method. Note that with only one frame there is no temporally encoded frame.

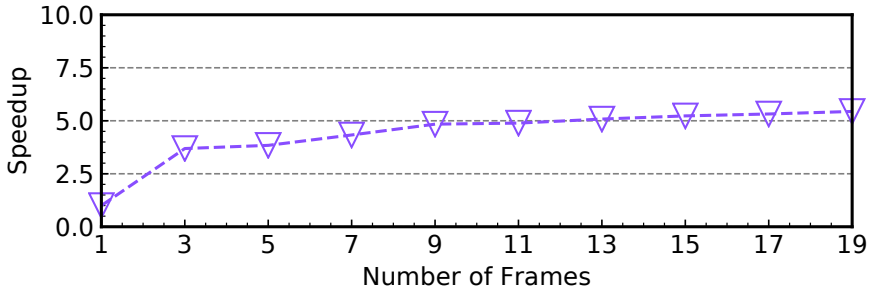


Fig. 7.12: Speedup of our parallel implementation over a sequential implementation on a four-core Intel i5-7500 CPU as the number of consecutive frames increases.

used (Huang et al., 2006; Kammerl et al., 2012; Smith et al., 2012; Hornung et al., 2013; Golla and Klein, 2015; Thanou et al., 2016; Lasserre et al., 2019). The G-PCC method in MPEG's point cloud compression standard falls into this category (Lasserre et al., 2019). Each Octree leaf node could be encoded by either a single occupancy bit, which could be lossless if each leaf node contains exactly one point, or by plane extraction, which preserves more details if each leaf node contains multiple points. G-PCC provides both options. Based on the space-partitioning tree representation, prior work has explored various methods to reduce redundant information, such as 2D pro-

jection (Hornung et al., 2013) or surface fitting (Smith et al., 2012).

Prior work also exploited temporal redundancies in space-partitioning trees such as XORing the two consecutive Octrees (Kammerl et al., 2012), using motion compensation in 3D space (Thanou et al., 2016), or applying video compression directly (Golla and Klein, 2015).

Other unstructured point cloud representations include shape adaptive wavelet (Ochotta and Saupe, 2004; Daribo et al., 2011) and hierarchical height map (Hornung et al., 2013; Morell et al., 2014). While effective in certain use cases, the downside of unstructured representations is that they do not exploit the unique characteristics exposed by LiDAR point clouds, leading to a low compression rate.

Structured Point Cloud Compression Instead of encoding point clouds using space-partitioning trees, another category of compression methods convert point clouds into 2D images using spherical projection (Tu et al., 2016, 2019; Sun et al., 2019) or orthogonal projection such as the V-PCC method in the MPEG's standard (Jang et al., 2019; Krivokuća et al., 2020). Existing image/video compression methods are then used to further compress the projected images (Jang et al., 2019; Tu et al., 2016; Sun et al., 2019). However, directly applying image/video compression algorithms does not preserve the spatial information inherit in the point cloud, and thus generally results in low application accuracy.

8 Mesorasi: Addressing Compute Inefficiencies via Delayed-Aggregation

Sec. 6.2 presents the irregular computation characteristics in deep point cloud analytics. Such computation characteristics also pose a tremendous challenge to hardware acceleration. This chapter proposes a technique called delayed-aggregation which can effectively compensate for irregular computations. Sec. 8.1 explains the insight of delay-aggregation technique. Sec. 8.2 proposes corresponding hardware supports for delay-aggregation. Next, Sec. 8.3 and Sec. 8.4 quantify the performance of delay-aggregation against its corresponding baseline. Finally, Sec. 8.5 highlights related work.

8.1 Delayed-Aggregation Algorithm

We introduce delayed-aggregation, a primitive for building efficient point cloud networks (Sec. 8.1.1). Delayed-aggregation improves the compute and memory efficiencies of point cloud networks without degrading accuracy (Sec. 8.1.2). We show that aggregation emerges as a new bottleneck in new networks, motivating dedicated hardware support (Sec. 8.1.3).

8.1.1 Algorithm

We propose a new framework for building efficient point cloud algorithms. The central idea is to delay aggregation until *after* feature computation so that features are extracted on individual input points rather than on aggregated neighbors. Delayed-aggregation has two benefits. First, it allows neighbor search and feature computation, the two time-consuming components, to be executed in parallel. Second, feature computation operates on input points rather than aggregated neighbors, reducing the compute and memory costs.

Delayed-Aggregation The key insight is that feature extraction ($\mathcal{F}$) is *approximately distributive* over aggregation ($\mathcal{A}$). For an input point $\mathbf{p}_i$ and its corresponding output $\mathbf{p}_o$:

$$\mathbf{p}_o = \mathcal{F}(\mathcal{A}(\mathcal{N}(\mathbf{p}_i),\ \ \mathbf{p}_i)) \approx \mathcal{A}(\mathcal{F}(\mathcal{N}(\mathbf{p}_i)),\ \mathcal{F}(\mathbf{p}_i)) \tag{8.1}$$

Fundamentally, Equ. 8.1 holds because the multi-layer perception (MLP) in $\mathcal{F}$ is approximately distributive over subtraction in $\mathcal{A}$. Specifically, applying an MLP to the difference between two matrices is approximately equivalent to applying an MLP to both matrices and then subtracting the two resulting matrices. The approximation is introduced by the non-linearity in the MLP (e.g., ReLU):

$$\begin{aligned} &\phi(\phi(\begin{bmatrix} \mathbf{p}_1 - \mathbf{p}_i \\ \dots \\ \mathbf{p}_k - \mathbf{p}_i \end{bmatrix} \times W_1) \times W_2) \approx \\ &\phi(\phi(\begin{bmatrix} \mathbf{p}_1 \\ \dots \\ \mathbf{p}_k \end{bmatrix} \times W_1 \times W_2)) - \phi(\phi(\begin{bmatrix} \mathbf{p}_i \\ \dots \\ \mathbf{p}_i \end{bmatrix} \times W_1 \times W_2)) \end{aligned} \tag{8.2}$$

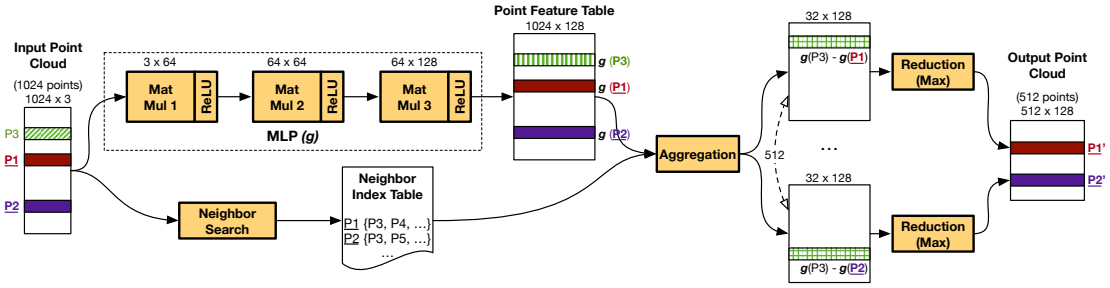


Fig. 8.1: The delayed-aggregation algorithm applied to the first module in PointNet++. The MLP and neighbor search are executed in parallel, effectively delaying aggregation after feature computation. The input size of the MLP is much smaller (input point cloud as opposed to the aggregated NFMs), which significantly reduces the MAC operations and the intermediate activation sizes. Aggregation now operates on the output feature space (128-D in this case), whereas it previously operates on the input feature space (3-D in this case). Thus, the aggregation time increases and emerges as a new performance bottleneck.

where $\mathbf{p}_1, ..., \mathbf{p}_k$ are neighbors of $\mathbf{p}_i$, W_1 and W_2 are the two weight matrices in the MLP (assuming one hidden layer), and ϕ is the non-linear activation function. Without ϕ, the distribution of MLP over subtraction is precise. In actual implementation, the computation on $[\mathbf{p}_i \ \ ... \ \ \mathbf{p}_i]^\intercal$ is simplified to operating on $\mathbf{p}_i$ once and scattering the result K times.

Critically, applying this distribution allows us to decouple $\mathcal{N}$ with $\mathcal{F}$. As shown in Equ. 8.1 and Equ. 8.2, $\mathcal{F}$ now operates on original input points, i.e., $\mathbf{p}_i$ and $\mathcal{N}(\mathbf{p}_i)$ (a subset of the input points, too) rather than the normalized point values $(\mathbf{p}_k - \mathbf{p}_i)$, which requires neighbor search results. As a result, we could first apply feature computation on all input points. The computed features are then aggregated later.

Walk-Through We use the first module in PointNet++ as an example to walk through the new algorithm. This module consumes 1024 (N_{in}) input points, among which 512 undergo neighbor search. Thus, the module produces 512 (N_{out}) output points. The input feature dimension is 3 (M_{in}) and the output feature dimension is 128

(M_{out}). Fig. 8.1 shows this module implemented with delayed-aggregation.

We first compute features ($\mathcal{F}$) from all 1024 points in the input point cloud and store the results in the Point Feature Table (PFT), a 1024×128 matrix. Every PFT entry contains the feature vector of an input point. Meanwhile, neighbor searches ($\mathcal{N}$) are executed in parallel on the input point cloud, each returning 32 neighbors of a centroid. The results of neighbor search are stored in a Neighbor Index Table (NIT), a 512×32 matrix. Each NIT entry contains the neighbor indices of an input point. In the end, the aggregation operation ($\mathcal{A}$) aggregates features in the PFT using the neighbor information in the NIT. Note that it is the features that are being aggregated, not the original points.

Each aggregated matrix (32×128) is reduced to the final feature vector (1×128) of an output point. If the reduction is implemented by a max operation as is the common case, aggregation could further be delayed after reduction because subtraction is distributive over max: $max(\mathbf{p}_1 - \mathbf{p}_i, \mathbf{p}_2 - \mathbf{p}_i) = max(\mathbf{p}_1, \mathbf{p}_2) - \mathbf{p}_i$. This optimization avoids scattering $\mathbf{p}_i$, reduces the subtraction cost, and is mathematically precise.

8.1.2 First-Order Efficiency Analysis

Compared with the original implementation of the same module in Fig. 6.3, the delayed-aggregation algorithm provides three benefits. First, neighbor search and the MLP are now executed in parallel, hiding the latencies of the slower path.

Second, we significantly reduce the MAC operations in the MLP. In this module, the original algorithm executes MLP on 512 32×3 matrices while the new algorithm executes MLP only on one 1024×3 matrix. Fig. 8.2 shows the MAC operation reductions across all five networks. On average, delayed-aggregation reduces the MAC counts by 68%.

Third, delayed-aggregation also reduces the memory traffic because the MLP input is much smaller. While the actual memory traffic reduction is tied to the hardware

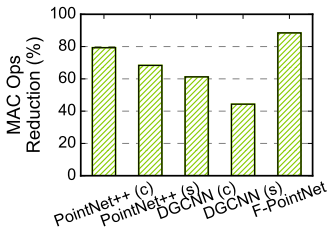


Fig. 8.2: MAC operation reduction in the MLP by delayed-aggregation. The MAC count reductions come from directly operating on the input points as opposed to aggregated neighbors.

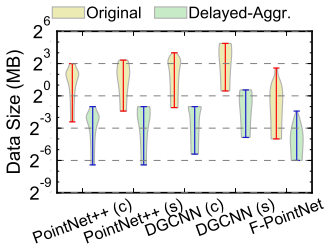


Fig. 8.3: Layer output size distribution as a violin plot with and without delayed-aggregation. High and low ticks denote the largest and smallest layer outputs, respectively.

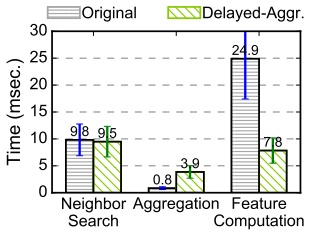


Fig. 8.4: Time distribution across $\mathcal{N}$, $\mathcal{A}$, and $\mathcal{F}$ in PointNet++ (s) with and without delayed-aggregation. Note that delayed-aggregation would also allow $\mathcal{N}$ and $\mathcal{F}$ to be executed in parallel.

architecture, as a first-order estimation Fig. 8.3 compares the distribution of per-layer output size with and without delayed-aggregation. The data is shown as a violin plot. Delayed-aggregation reduces the layer output sizes from 8 MB˜32 MB to 512 KB˜1 MB, amenable to be buffered completely on-chip.

By directly extracting features from the input points, our algorithm unlocks the inherent data reuse opportunities in point cloud. Specifically in this example, `P3` is a neighbor of both `P1` and `P2`, but could not be reused in feature computation by the original algorithm because `P3`'s normalized values with respect to `P1` and `P2` are different. In contrast, the MLP in our algorithm directly operates on `P1`, whose feature is then reused in aggregation, *implicitly* reusing `P1`.

8.1.3 Bottleneck Analysis

While delayed-aggregation reduces the compute costs and memory accesses, it also significantly increases the aggregation time. Using PointNet++ as an example, Fig. 8.4

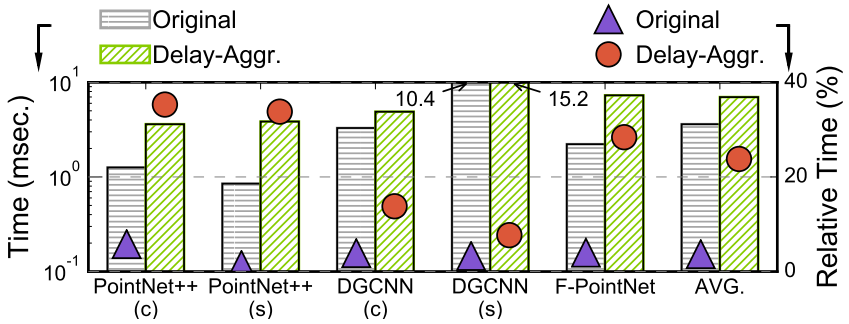


Fig. 8.5: Both absolute (left y-axis) and relative (right y-axis) aggregation times increase with delayed-aggregation.

compares the execution time distribution across the three operations ($\mathcal{N}$, $\mathcal{A}$, and $\mathcal{F}$) with and without delayed-aggregation. The error bars denote one standard deviation in the measurement. The feature extraction time significantly decreases, and the neighbor search time roughly stays the same — both are expected. The aggregation time, however, significantly increases.

Fig. 8.5 generalizes the conclusion across the five networks. The figure compares the absolute (left y-axis) and relative (right y-axis) aggregation time in the original and new algorithms. The aggregation time consistently increases in all five networks. Since neighbor search and feature computation are now executed in parallel, aggregation overhead contributes even more significantly to the overall execution time. On average, the aggregation time increases from 3% to 24%.

Aggregation time increases mainly because aggregation involves irregular gather operations (Kirk and Wen-Mei, 2016), which now operate on a much larger working set with delayed-aggregation. For instance, in PointNet++'s first module (Fig. 8.1), aggregation originally gathers from a 12 KB matrix but now gathers from a 512 KB matrix, which is much larger than the L1 cache size (48 KB – 96 KB[1]) in the mobile

[1]To our best knowledge, Nvidia does not publish the L1 cache size for the mobile Pascal GPU in TX2 (GP10B (Nvidia, 2017a)). We estimate the size based on the L1 cache size per SM in other Pascal GPU chips (Wikipedia, 2016) and the number of SMs in the mobile Pascal GPU (Nvidia, 2017b)

Pascal GPU on TX2.

The working set size increases significantly because aggregation in new algorithms gathers data from the PFT, whose dimension is $N_{in} \times M_{out}$, whereas the original algorithms gather data from the input point matrix, whose dimension is $N_{in} \times M_{in}$. M_{out} is usually several times greater than M_{in} in order to extract higher-dimensional features. In the example above, M_{out} is 128-D whereas M_{in} is 3-D.

8.2 Architectural Support

This section describes MESORASI, our hardware design that efficiently executes point cloud algorithms developed using delayed-aggregation. MESORASI extends existing DNN accelerators with minor augmentations while leaving the rest of the SoC untouched. We start from an overview of MESORASI and its workflow (Sec. 8.2.1), followed by a detailed description of the architecture support (Sec. 8.2.2).

8.2.1 Overall Design

We assume a baseline SoC that incorporates a GPU and an NPU, as with emerging mobile SoCs such as Nvidia Xavier (Nvidia, 2018), Apple A13 (Eadicicco, 2022), and Microsoft HPU (Microsoft, 2017). Point cloud algorithms are a few times faster when an NPU is available to accelerate MLP compared to running only on the GPU (Sec. 8.4.4). Thus, an NPU-enabled SoC represents the trend of the industry and is a more optimized baseline.

Design Fig. 8.6 shows how MESORASI augments the NPU in a generic SoC. In MESORASI, the GPU executes neighbor search ($\mathcal{N}$) and the NPU executes feature extraction ($\mathcal{F}$), i.e., the MLP. In addition, MESORASI augments the NPU with an Aggregation Unit (AU) to efficiently execute the aggregation operation ($\mathcal{A}$). As shown in Sec. 8.1.3, aggregation becomes a bottleneck in our new algorithms and is inefficient

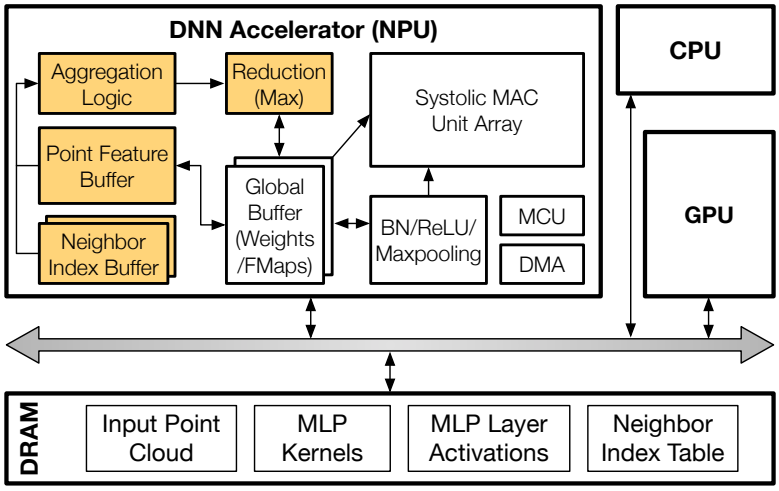


Fig. 8.6: The MESORASI SoC builds on top of today's SoCs consisting of a GPU and a DNN accelerator (NPU). Neighbor search executes on the GPU and feature extraction executes on the NPU. MESORASI augments the NPU with an aggregation unit (AU) to efficiently execute the aggregation operation. The AU structures are shaded (colored).

on the GPU. AU minimally extends a generic NPU architecture with a set of principled memory structures and datapaths.

MESORASI maps $\mathcal{N}$ to the GPU because neighbor search is highly parallel, but does not map to the specialized datapath of an NPU. Alternatively, an SoC could use a dedicated neighbor search engine (NSE) (Kuhara et al., 2013; Xu et al., 2019). We use the GPU because it is prevalent in today's SoCs and thus provides a concrete context to describe our design. We later show that delayed-aggregation could achieve even higher speedups in a futurist SoC where an NSE is available to accelerate neighbor search (Sec. 8.4.5). In either case, MESORASI does not modify the internals of the GPU or the NSE.

Work Flow Point cloud algorithms with delayed-aggregation work on MESORASI as follows. The input point cloud is initially stored in the DRAM. The CPU configures and triggers the GPU and the NPU simultaneously, both of which read the input point cloud. The GPU executes the KNN search and generates the Neighbor Index Table

(NIT), which gets stored back to the DRAM. Meanwhile, the NPU computes features for input points and generates the Point Feature Table (PFT). The AU in NPU combines the PFT with the NIT from the memory for aggregation and reduction, and eventually generates the output of the current module.

In some algorithms (e.g., PointNet++), neighbor searches in all modules search in the original 3-D coordinate space, while in other algorithms (e.g., DGCNN) the neighbor search in module i searches in the output feature space of module $(i-1)$. In the latter case, the current module's output is written back to the memory for the GPU to read in the next module.

Our design modifies only the NPU while leaving other SoC components untouched. This design maintains the modularity of existing SoCs, broadening the applicability. We now describe the AU augmentation in NPU in detail.

8.2.2 Aggregation Unit in NPU

Aggregation requires irregular gathering operations that are inefficient on GPUs. The key to our architectural support is the specialized memory structures co-designed with customized data structure partitioning, which provide efficient data accesses for aggregation with a little area overhead.

Algorithmically, aggregation iterates over the NIT's N_{out} entries until NIT is exhausted. Each NIT entry contains the K neighbor indices of a centroid $\mathbf{p}$. The aggregation operation first gathers the K corresponding entries (feature vectors) from the PFT ($N_{in} \times M_{out}$). The K feature vectors are then reduced to one ($1 \times M_{out}$) vector, which subtracts $\mathbf{p}$'s feature vector to generate the output feature for $\mathbf{p}$.

Fig. 8.7 shows the detailed design of the aggregation unit. The NIT is stored in an SRAM, which is doubled-buffered in order to limit the on-chip memory size. The PFT is stored in a separate on-chip SRAM connected to the NPU's global buffer (which stores the MLP weights and input/output). This allows the output of feature extraction

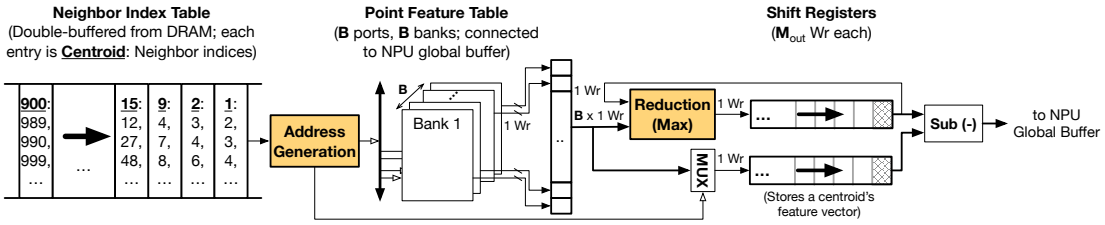


Fig. 8.7: Aggregation unit. The NIT buffer is double-buffered from the DRAM. The Address Generation logic simply fetches addresses already buffered in the NIT and sends them to the PFT buffer controller. The PFT buffer is organized as B independently addressed single-ported SRAMs. It could be thought of as an optimized version of a traditional B-banked, B-ported SRAM, because it does not need the crossbar that routes data from banks to ports (but does need the crossbar to route an address to the corresponding bank). The PFT buffer is connected to the NPU's global buffer. Each bank produces one word (Wr) per cycle. The shift registers hold up to M_{out} words, where M_{out} is the output feature vector size. The top shift register holds the result of the reduction, and the bottom shift register holds the feature vector of a centroid.

to be directly transferred to the PFT buffer without going through the DRAM. Similarly, the aggregation output is directly written back to the NPU's global buffer, as the aggregation output of the current module is the input to the feature extraction in the next module.

To process each NIT entry, the Address Generation Unit (AGU) uses the K indices to generate K addresses to index into the PFT buffer. Due to the large read bandwidth requirement, the PFT buffer is divided into B independently addressable banks, each of which produces 1 word per cycle.

Each cycle, the PFT buffer produces B words, which enters the reduction unit. In our current design, the reduction unit implements the max logic as is the case in today's point cloud algorithms. The output of the max unit, i.e., the max of the B words, enters a shift register (the top one in Fig. 8.7). Ideally, the number of banks B is the same as the number of neighbors K and the K addresses fall into different banks. If so, the

shift register is populated with the $1 \times M_{out}$ vector after M_{out} cycles. The AGU then reads $\mathbf{p}$'s feature vector from the PFT buffer and stores it in another shift register (the bottom one in Fig. 8.7). The two shift registers perform an element-wise subtraction as required by aggregation. The same process continues until the entire NIT is exhausted.

Multi-Round Grouping In reality, reading the neighbor feature vectors takes more than M_{out} cycles because of two reasons. First, K could be greater than B. The number of banks B is limited by the peripheral circuits overhead, which increases as B increases. Second, some of the K addresses could fall into the same bank, causing bank conflicts. We empirically find that an LSB-interleaving reduces bank conflicts, but it is impossible to completely avoid bank conflict at runtime, because the data access patterns in point cloud are irregular and could not be statically calculated – unlike conventional DNNs and other regular kernels.

We use a simple multi-round design to handle both non-ideal scenarios. Each round the AGU would attempt to identify as many unconflicted addresses as possible, which is achieved by the AGU logic examining each address modulo B. The unconflicted addresses are issued to the PFT buffer, whose output enters the max unit to generate a temporary output stored in the shift register. The data in the shift register would be combined with the PFT output in the next round for reduction. This process continues until all the addresses in an NIT buffer entry are processed.

An alternative way to resolve bank-conflict would be to simply ignore conflicted banks, essentially approximating the aggregation operation. We leave it to future work to explore this optimization and its impact on the overall accuracy.

PFT Buffer Design One could think of the PFT buffer as a B-banked, B-ported SRAM. Traditionally, heavily ported and banked SRAMs are area inefficient due to the crossbar that routes each bank's output to the corresponding issuing port (Weste and Harris, 2015). However, our PFT buffer is much simplified *without* the crossbar. This is by leveraging a key observation that the outputs of all the PFT banks are consumed by the max unit, which executes a *commutative* operation, i.e., $max(a, b) = max(b, a)$.

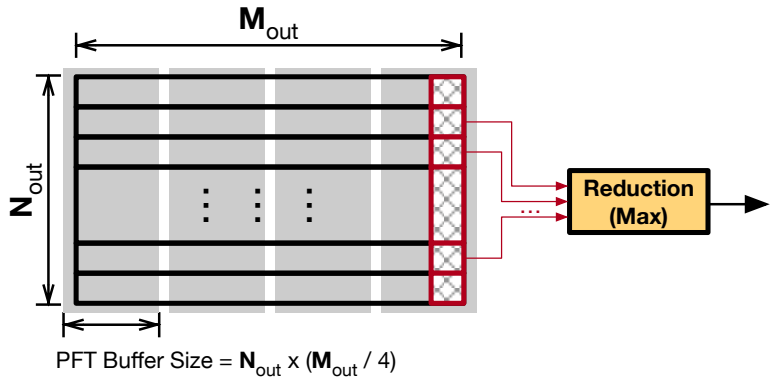


Fig. 8.8: Column-major partitioning of PFT to reduce PFT buffer size (4 partitions in this example). Each time the PFT buffer is filled with only one partition. Since reduction (max) is applied to each column independently, the column-major partitioning ensures that all the neighbors of a centroid are present in the PFT buffer for aggregation.

Thus, the output of each bank need not be routed to the issuing port so long as the requested data is correctly produced. This design optimization greatly reduces the area overhead (Sec. 8.4.1).

One might be tempted to reuse the NPU's global buffer for the PFT buffer to save chip area. After all, the PFT *is* MLP's output, which is stored in the global buffer. However, physically sharing the two SRAM structures is difficult, mainly because of their different design requirements. Global buffer contains MLP weights and layer inputs, accesses to which have regular patterns. As a result, NPU global buffers are usually designed with very few ports (e.g., one) (Jouppi et al., 2017b; Arm, 2018) while using a wide word. In contrast, accesses to the PFT are irregular as the neighbors of a centroid could be arbitrary spread in the PFT. Thus, the PFT buffer must be heavily-ported in order to sustain a high bandwidth requirement.

PFT Partitioning To hold the entire PFT, the buffer must hold $N_{out} \times M_{out}$ features, which could be as large as 0.75 MB in some networks (e.g., DGCNN). Since the PFT buffer adds area overhead, we would like to minimize its size.

We partition the PFT to reduce the PFT buffer size. Each time, the PFT buffer is filled with only one partition. One straightforward strategy is the row-major partitioning, where the PFT buffer holds only a few rows of the PFT. However, since a centroid's neighbors could be arbitrarily spread across different PFT rows, row-major partitioning does not guarantee that all the neighbors of a centroid are present in the PFT buffer (i.e., in the same partition) for aggregation.

Instead, our design partitions the PFT column-wise, where each partition contains several columns of the PFT. Fig. 8.8 illustrates the idea with 4 partitions. In this way, the aggregation of a centroid is divided into four steps, each step aggregating only one partition. The column-major partitioning ensures that, within each partition, the neighbors of a centroid are available in the PFT buffer. Since reductions (max) of different columns are independent, the four intermediate reduction results can simply be concatenated in the end.

With column-wise partitioning, each NIT entry is accessed multiple times—once per aggregation step. Thus, a smaller PFT buffer, while reducing the area overhead, would also increase the energy overhead. We later quantify this resource vs. energy trade-off (Sec. 8.4.6).

8.3 Experimental Setup

Hardware Implementation We develop RTL implementations for the NPU and its augmentations for the aggregation unit (AU). The NPU is based on the systolic array architecture, and consists of a 16×16 PE array. Each PE consists of two input registers, a MAC unit with an accumulator register, and simple trivial control logic. This is identical to the PE in the TPU (Jouppi et al., 2017b). Recall that MLPs in point cloud networks process batched inputs (Fig. 6.3), so the MLPs perform matrix-matrix product that can be efficiently implemented on a systolic array. The NPU's global buffer is sized at 1.5 MB and is banked at a 128 KB granularity.

The PFT buffer in the AU is sized at 64 KB with 32 banks. The NIT buffer is doubled-buffered; each buffer is implemented as one SRAM bank sized at 12 KB and holds 128 entries. The NIT buffer produces one entry per cycle. Each entry is 98 Bytes, accommodating 64 neighbor indices (12 bits each). Each of the two shift registers is implemented as 256 flip-flops (4-byte each). The datapath mainly consists of 1) one 33-input max unit and 256 subtraction units in the reduction unit, and 2) 32 32-input MUXes in the AGU.

The design is clocked at 1 GHz. The RTL is implemented using Synposys synthesis and Cadence layout tools in TSMC 16nm FinFET technology, with SRAMs generated by an Arm memory compiler. Power is simulated using Synopsys PrimeTimePX, with fully annotated switching activity.

Experimental Methodology The latency and energy of the NPU (and its augmentation) are obtained from the post-synthesis results of the RTL design. We model the GPU after the Pascal mobile GPU in the Nvidia Parker SoC hosted on the Jetson TX2 development board (Nvidia, 2017b). The SoC is fabricated in a 16 nm technology node, same as our NPU. We directly measure the GPU execution time as well as the kernel launch time. The GPU energy is directly measured using the built-in power sensing circuity on TX2.

The DRAM parameters are modeled after Micron 16 Gb LPDDR3-1600 (4 channels) according to its datasheet (Micron, 2014). DRAM energy is calculated using Micron's System Power Calculators (Micron, 2020) using the memory traffic, which includes: 1) GPU reading input point cloud, 2) NPU accessing MLP kernels and activations each layer, and 3) GPU writing NIT and NPU reading NIT. Overall, the DRAM energy per bit is about 70× of that of SRAM, matching prior work (Yazdanbakhsh et al., 2018; Gao et al., 2017).

The system energy is the aggregation of GPU, NPU, and DRAM. The overall latency is the sum of GPU, NPU, and DRAM minus: 1) double buffering in the NPU, and 2) parallel execution between neighbor search on GPU and feature computation on

Table 8.1: Evaluation benchmarks.

Application Domains	Algorithm	Dataset	Year
Classification	PointNet++ (c), DGCNN (c), LDGCNN, DensePoint	ModelNet40	2017, 2019, 2019, 2019
Segmentation	PointNet++ (s), DGCNN (s)	ShapeNet	2017, 2019
Detection	F-PointNet	KITTI	2018

NPU. Due to double-buffering, the overall latency is dominated by the compute latency, not memory.

Software Setup Tbl. 8.1 lists the point cloud networks we use, which cover different domains for point cloud analytics including object classification, segmentation, and detection. The networks cover both classic and recent ones (2019).

For classification, we use four networks: PointNet++ (Qi et al., 2017b), DGCNN (Wang et al., 2019a), LDGCNN (Zhang et al., 2019a), and DensePoint (Liu et al., 2019b); we use the ModelNet40 (Wu et al., 2015) dataset. We report the standard overall accuracy metric. To evaluate segmentation, we use the variants of PointNet++ and DGCNN specifically built for segmentation, and use the ShapeNet dataset (Chang et al., 2015). We report the standard mean Intersection-over-Unit (mIoU) accuracy metric. Finally, we use F-PointNet (Qi et al., 2018) as the object detection network. We use the KITTI dataset (Geiger et al., 2012a) and report the geometric mean of the IoU metric (BEV) across its classes.

We optimize the author-released open-source version of these networks to obtain stronger software baselines. We: 1) removed redundant data duplications introduced by `tf.tile`; 2) accelerated the CPU implementation of an important kernel, 3D Interpretation (`three_interpolate`), with a GPU implementation; 3) replaced the Farthest Point Sampling with random sampling in PointNet++ with little accuracy loss; 4) replaced the Grouping operation (`group_point`) with an optimized implementation (`tf.gather`) to improve the efficiency of grouping/aggregation. On TX2, our

baseline networks are 2.2× faster than the open-source versions.

Baseline We mainly compare with a generic NPU+GPU SoC without any MESORASI-related optimizations. Compared to the baseline, our proposal improves both the software, i.e., the delayed-aggregation algorithm as well as hardware, i.e., the aggregation unit (AU) augmentations to the NPU.

Variants To decouple the contributions of our algorithm and hardware, we present two different MESORASI variants:

- MESORASI-SW: delayed-aggregation without AU support. Neighbor search and aggregation execute on the GPU; feature computation executes on the NPU.
- MESORASI-HW: delayed-aggregation with AU support. Neighbor search executes on the GPU; aggregation and feature computation execute on the NPU.

8.4 Evaluation

We first show MESORASI adds little hardware overhead (Sec. 8.4.1) while achieving comparable accuracy against original point cloud networks (Sec. 8.4.2). We then demonstrate the efficiency gains of MESORASI on different hardware platforms (Sec. 8.4.3 – Sec. 8.4.5), followed by sensitivity studies (Sec. 8.4.6).

8.4.1 Area Overhead

MESORASI introduces only minimal area overhead with the minor AU augmentations. The main overhead comes from the 88 KB additional SRAM required for the PFT buffer and the NIT buffer. Compared to the baseline NPU, the additional hardware introduces less than 3.8% area overhead ($0.059\,\mathrm{mm}^2$), which is even more negligible compared to the entire SoC area (e.g., $350\,\mathrm{mm}^2$ for Nvidia Xavier (Schor, 2018) and $99\,\mathrm{mm}^2$ for Apple A13 (Eadicicco, 2022)).

Our custom-designed PFT buffer avoids the crossbar connecting the banks to the read ports by exploiting the algorithmic characteristics (Sec. 8.2.2). Since our PFT buffer is heavily banked (32) and ported (32) and each bank is small in size (2 KB), the additional area overhead introduced by the crossbar would have been high. Specifically, the area of the PFT buffer now is $0.031\,\mathrm{mm}^2$, but the crossbar area would be $0.064\,\mathrm{mm}^2$, which is now avoided.

8.4.2 Accuracy

Overall, MESORASI matches or out-performs the original algorithms. We train all seven networks with delayed-aggregation from scratch until the accuracy converges. Fig. 8.9 compares our accuracy with that of the baseline models, which we choose the better of the reported accuracies in the original papers or accuracies from training their released code. Overall, MESORASI leads to at most 0.9% accuracy loss in the case of PointNet++ (c) and up to 1.2% accuracy gain in the case of F-PointNet. This shows that, while delayed-aggregation approximates the original algorithms, the accuracy loss could be recovered from training. Delayed-aggregation could be used as a primitive to build accurate point cloud algorithms.

We find that fine-tuning the model weights trained on the original networks has similar accuracies as retraining from scratch. However, directly using the original weights without retraining leads to a few percentages of accuracy loss, which is more significant when the non-linear layers use batch normalization, which perturbs the distributive property of matrix multiplication over subtraction more than ReLU.

8.4.3 Results on GPU

We first show that our delayed-aggregation algorithm readily achieves significant speedups on today's GPU *without* hardware support. Fig. 8.10 shows the speedup and energy reduction of MESORASI on the Pascal GPU on TX2.

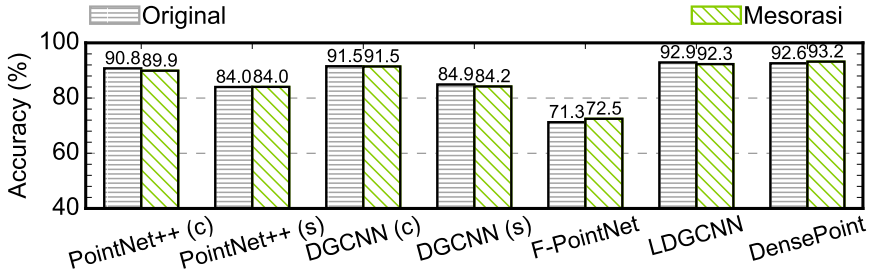


Fig. 8.9: The accuracy comparison between networks trained with delayed-aggregation and the original networks.

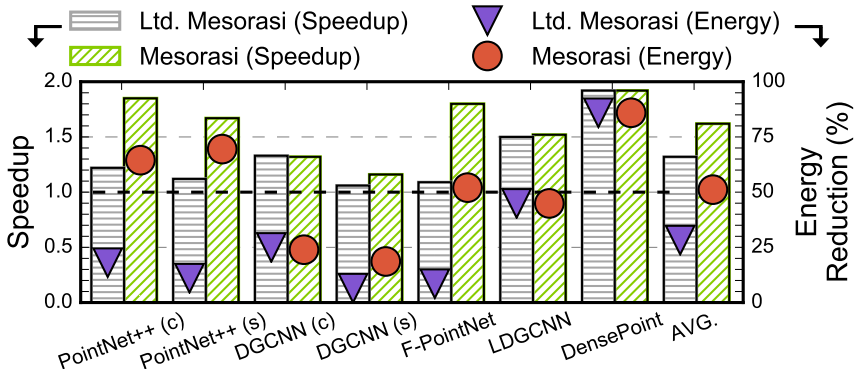


Fig. 8.10: Speedup and energy reduction of the delayed-aggregation algorithm and the limited version of the algorithm on the mobile Pascal GPU on TX2.

As a comparison, we also show the results of a limited version of delayed-aggregation, where only the matrix-vector multiplication (MVM) part of an MLP is hoisted before aggregation (Ltd-MESORASI). The limited delayed-aggregation algorithm is inspired by certain Graph Neural Network (GNN) implementations such as GCN (Fey and Lenssen, 2019; Wang et al., 2019b) and GraphSage (Hamilton, 2017). Note that by hoisting only the MVM rather than the entire MLP, Ltd-MESORASI is precise since MVM is linear.

On average, MESORASI achieves $1.6\times$ speedup and 51.1% energy reduction compared to the original algorithms. In comparison, the limited delayed-aggregation algorithm achieves only $1.3\times$ speedup and 28.3% energy reduction. Directly comparing with Ltd-MESORASI, MESORASI has $1.3\times$ speedup and 25.9% energy reduction. This

is because the limited delay-aggregation, in order to be precise, could be applied to only the first MLP layer. By being approximate, MESORASI does not have this constraint and thus enables larger benefits; the accuracy loss could be recovered through fine-tuning (Fig. 8.9). MESORASI has similar performance as Ltd-MESORASI on DGCNN (c), LDGCNN, and DensePoint, because these three networks have only one MLP layer per module.

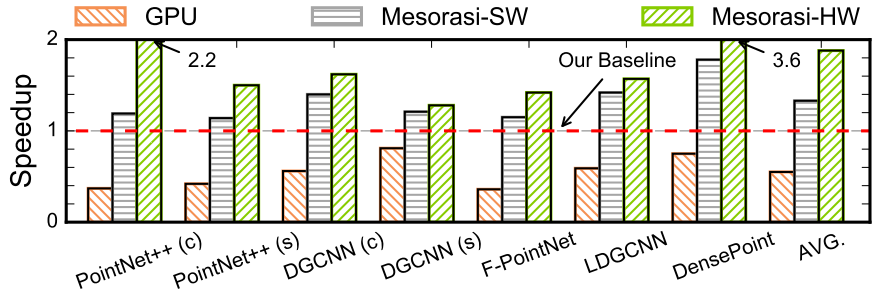
Although delayed-aggregation allows neighbor search and feature extraction to be executed in parallel, and our implementation does exploit the concurrent kernel execution in CUDA, we find that neighbor search and feature extraction in actual executions are rarely overlapped. Further investigation shows that this is because the available resources on the Pascal GPU on TX2 are not sufficient to allow both kernels to execute concurrently. We expect the speedup to be even higher on more powerful mobile GPUs in the future.

Overall, networks in which feature computation contributes more heavily to the overall time, such as PointNet++ (c) and F-PointNet (Fig. 6.5), have higher MAC operation reductions (Fig. 8.2), and thus have higher speedups and energy reductions. This confirms that the improvements are mainly attributed to optimizing the MLPs in feature computation.

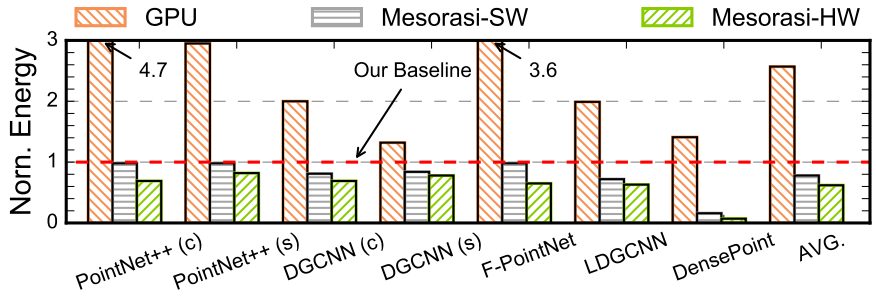
8.4.4 Speedup and Energy Reduction

MESORASI also improves the performance and energy consumption of emerging mobile SoCs with a dedicated NPU. Fig. 8.11a and Fig. 8.11b show the speedup and the normalized energy consumption of the two MESORASI variants over the NPU+GPU baseline, respectively.

Software The delayed-aggregation algorithm alone without AU support, i.e., MESORASI-SW, has a $1.3\times$ speedup and 22% energy saving over the baseline. The main contributor of the improvements is optimizing the MLPs in feature computation.



(a) Speedup. Higher is better.



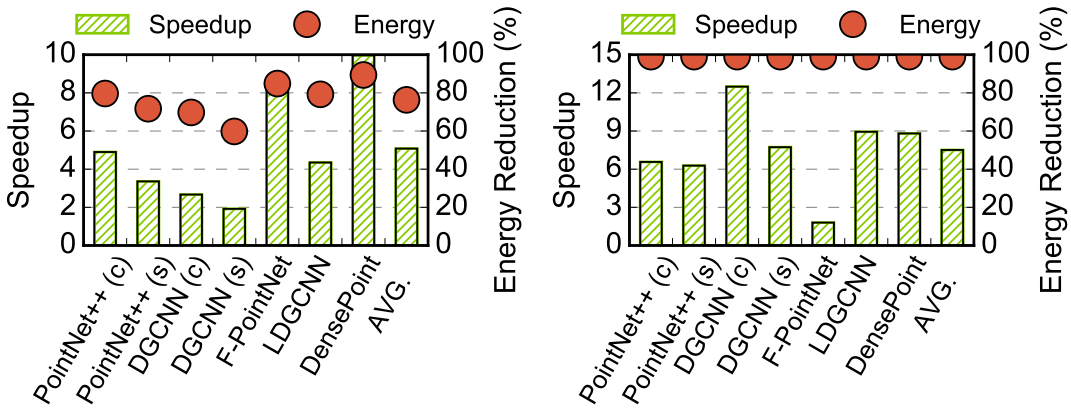
(b) Normalized energy. Lower is better.

Fig. 8.11: Speedup and energy reduction of MESORASI-SW and MESORASI-HW over the baseline (GPU+NPU), which is twice as fast and consumes one-third of the energy compared to the GPU, indicating an optimized baseline to begin with.

Fig. 8.12a shows the speedups and energy savings of the delayed-aggregation algorithm on feature computation. On average, the feature computation time is reduced by $5.1\times$ and the energy consumption is reduced by 76.3%.

The large speedup on feature computation does not translate to similar overall speedup, because feature computation time has already been significantly reduced by the NPU, leaving less room for improvement. In fact, our GPU+NPU baseline is about $1.8\times$ faster (Fig. 8.11a) and consumes 70% less energy compared to the GPU (Fig. 8.11b). The increased workload of aggregation also adds to the overhead, leading to overall lower speedup and energy reduction than on GPU.

Hardware With the AU hardware, MESORASI-HW boosts the speedup to $1.9\times$ (up to $3.6\times$) and reduces the energy consumption by 37.6% (up to 92.9%). DGCNN (s)



(a) Feature computation.

(b) Aggregation.

Fig. 8.12: Speedup and energy savings on feature computation and aggregation.

has the least speedup because it has the least aggregation time (Fig. 8.5), thus benefiting the least from the AU hardware.

Fig. 8.12 shows the speedup and energy reduction of aggregation over the baseline (which executes aggregation on the GPU). Overall, MESORASI-HW reduces the aggregation time by 7.5× and reduces the energy by 99.4%. The huge improvements mainly come from using a small memory structure customized to the data access patterns in aggregation.

On average, 27% (max 29%) of PFT buffer accesses are to serve previous bank conflicts. The total time spent on PFT buffer accesses is 1.5× of the ideal case without bank conflicts. Empirically we do not observe pathological cases.

The AU's speedup varies across networks. For instance, the speedup on PointNet++ (c) is over 3× higher than that of F-PointNet. This is because the speedup decreases as bank conflict increases; bank conflicts occur more often when neighbor search returns more neighbors. The neighbor searches in PointNet++ (c) mostly return 32 neighbors, whereas neighbor searches in F-PointNet return mostly 128 neighbors, significantly increasing the chances of bank conflicts. This also explains why PointNet++ (c) has overall higher speedup than F-PointNet (Fig. 8.11a).

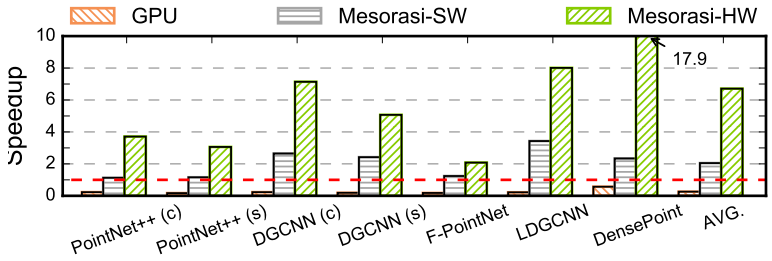


Fig. 8.13: MESORASI-SW and MESORASI-HW speedup over an NSE-enabled SoC (GPU+NPU+NSE), which is 4.0× faster than the GPU by accelerating both MLP and neighbor search.

8.4.5 Results with Neighbor Search Engine (NSE)

From the evaluations above, it is clear that the improvements of MESORASI will ultimately be limited by the neighbor search overhead, which MESORASI does not optimize and becomes the "Amdahl's law bottleneck."

To assess the full potential of MESORASI, we evaluate it in a futuristic SoC that incorporates a dedicated neighbor search engine (NSE) that accelerates neighbor searches. We implement a recently published NSE built specifically for accelerating neighbor searches in point cloud algorithms (Xu et al., 2019), and incorporate it into our SoC model. On average, the NSE provides over 60× speedup over the GPU. Note that the NSE is *not* our contribution. Instead, we evaluate the potential speedup of MESORASI if an NSE is available.

The speedup of MESORASI greatly improves when neighbor search is no longer a bottleneck. Fig. 8.13 shows the speedups of MESORASI-SW and MESORASI-HW on the NSE-enabled SoC. On average, MESORASI-SW achieves 2.1× speedup and MESORASI-HW achieves 6.7× speedup. The two DGCNN networks have particularly high speedups because neighbor search contributes heavily to their execution times (Fig. 6.5).

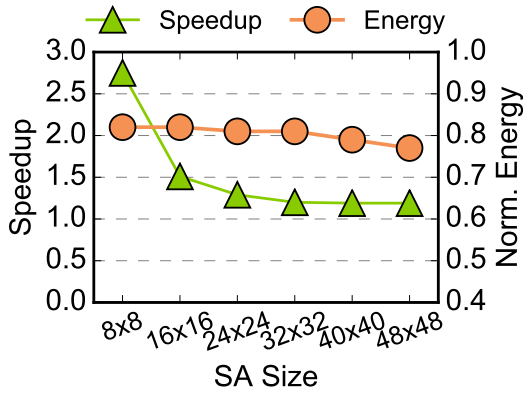


Fig. 8.14: Sensitivity of the speedup and energy to the systolic array size.

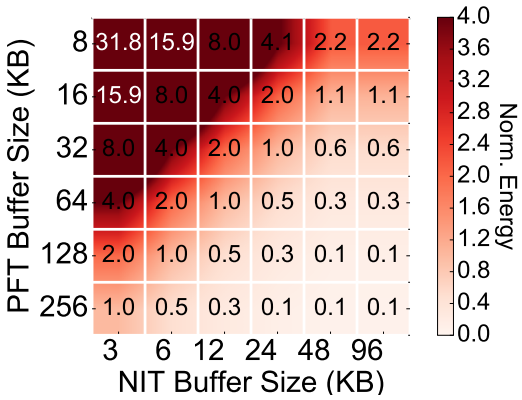


Fig. 8.15: Sensitivity of AU energy consumption to the NIT/PFT buffer sizes.

8.4.6 Sensitivity Study

The evaluations so far are based on one hardware configuration. We now study how the improvements vary with different hardware resource provisions. In particular, we focus on two types of hardware resources: the baseline NPU and the AU augmentation. Due to the page limit, we show only the results of PointNet++ (s). The general trend holds.

NPU We find that MESORASI has higher speedups when the NPU is smaller. Fig. 8.14 shows how the speedup and normalized energy of MESORASI-HW over the baseline vary as the systolic array (SA) size increases from 8×8 to 48×48. As the SA size increases, the feature extraction time decreases, and thus leaving less room for performance improvement. As a result, the speedup decreases from 2.8× to 1.2×.

Meanwhile, the energy reduction increases from 17.7% to 23.4%. This is because a large SA is more likely throttled by memory bandwidth, leading to overall higher energy.

AU We find that the AU energy consumption is sensitive to the NIT and PFT buffer sizes. Fig. 8.15 shows the AU energy under different NIT and PFT buffer sizes. The results are normalized to the nominal design point described in Sec. 8.3 (i.e., a 64 KB of PFT and a 12 KB NIT).

The energy consumption increases as the PFT and NIT buffer sizes decrease. In an extremely small setting with an 8 KB PFT buffer and a 3 KB NIT buffer, the AU energy increases by 32×, which leads to a 5.6% overall energy increase. This is because a smaller PFT buffer leads to more PFT partitions, which increases NIT buffer energy since each NIT entry must be read once per partition. Meanwhile, a smaller NIT requires more DRAM accesses, whose energy dominates as the PFT buffer becomes very small. On the other extreme, using a 256 KB PFT buffer and a 96 KB NIT buffer reduces the overall energy by 2.0% while increasing the area overhead by 4×. Our design point balances energy saving and area overhead.

8.5 Related Work

Point Cloud Analytics Point cloud has only recently received extensive interests. Unlike conventional image/video analytics, point cloud analytics requires considerably different algorithms due to the unique characteristics of point cloud data. Most of the recent work focuses on the accuracy, exploring not only different data representation (e.g., 2D projection (Qi et al., 2016), voxelization (Wu et al., 2015; Riegler et al., 2017), and raw points (Qi et al., 2017b; Wang et al., 2019a)), but also different ways to extract features from points (Simonovsky and Komodakis, 2017; Qi et al., 2017b; Wu et al., 2019; Wang et al., 2019a). Our delayed-aggregation primitive can be thought of as a new, and efficient, way of extracting features from raw points.

MESORASI focuses on improving the efficiency of point cloud algorithms while retaining the high accuracy. In the same vein, PVCNN (Liu et al., 2019c) combines point-based and voxel-based data representations in order to boost compute and memory efficiency. Different but complementary, MESORASI focuses on point-based neural networks. While PVCNN is demonstrated on GPUs, MESORASI not only directly benefits commodity GPUs, but also incorporates systematic hardware support that improves DNN accelerators.

Prior work has also extensively studied systems and architectures for accelerating neighbor search on GPU (Qiu et al., 2009; Gieseke et al., 2014), FPGA (Heinzle et al., 2008; Winterstein et al., 2013; Kuhara et al., 2013), and ASIC (Xu et al., 2019). Neighbor search contributes non-trivial execution time to point cloud networks. MESORASI hides, rather than reduces, the neighbor search latency, and directly benefits from faster neighbor search.

GNNs Point cloud applications bear some resemblance to GNNs. After all, both deal with spatial/geometric data. In fact, some point cloud applications are implemented using GNNs, e.g., DGCNN (Wang et al., 2019a).

However, existing GNN accelerators, e.g., HyGCN (Yan et al., 2020), are insufficient in accelerating point cloud applications. Fundamentally, GNN does not require explicit neighbor search (as a vertex's neighbors are explicitly encoded), but neighbor search is a critical bottleneck of all point cloud applications, as points are arbitrarily spread in 3D space. Our design hides the neighbor search latency, which existing GNN accelerators simply do not optimize for. In addition, MESORASI minimally extends conventional DNN accelerators instead of being a new accelerator design, broadening its applicability in practice.

From Fig. 6.5, one might notice that $\mathcal{A}$ in point cloud networks is much faster than $\mathcal{F}$, which is the opposite in many GNNs (Yan et al., 2020). This is because $\mathcal{F}$ in point cloud applications does much more work than $\mathcal{A}$, opposite to GNNs. In point cloud application, $\mathcal{A}$ simply gathers neighbor feature vectors, and $\mathcal{F}$ operates on neighbor feature vectors (MLP on each vector). In contrast, $\mathcal{A}$ in GNNs gathers and reduces neighbor feature vectors to one vector, and $\mathcal{F}$ operates on the reduced vector (MLP on one vector).

Domain-Specific Accelerator Complementary to improving generic DNN accelerators, much of recent work has focused on improving the DNN accelerators for specific application domains such as real-time computer vision (Buckler et al., 2018; Zhu et al., 2018b; Feng et al., 2019), computational imaging (Mahmoud et al., 2018;

Huang et al., 2019), and language processing (Riera et al., 2018). The NPU in the MESORASI architecture is a DNN accelerator specialized to point cloud processing. MESORASI also extends beyond prior visual accelerators that deal with 2D data (images and videos) (Mahmoud et al., 2017; Zhang et al., 2017; Mazumdar et al., 2017; Zhang et al., 2019b; Leng et al., 2019; Zhao et al., 2020) to 3D point clouds.

To keep the modularity of existing SoCs, MESORASI relies on the DRAM for inter-accelerator communication. That said, MESORASI could benefit from more direct accelerator communication schemes such as VIP (Nachiappan et al., 2016) and Short-circuiting (Yedlapalli et al., 2014). For instance, the NIT could be directly communicated to the NIT buffer from the GPU through a dedicated on-chip link, pipelining neighbor search with aggregation.

9 Crescent: Addressing Memory Inefficiencies via Compulsory Approximation

Irregular memory accesses plague point cloud analytics and introduce massive memory inefficiencies. To address irregular memory accesses, this chapter exploits the approximate nature of deep learning and proposes an approximation technique to alleviate memory irregularity. Sec. 9.1 and Sec. 9.2 propose techniques to address memory inefficiencies in DRAM and SRAM, respectively. Sec. 9.3 introduces a co-training strategy to effectively train such an approximation algorithm. The quantitative evaluation is shown in Sec. 9.4 and Sec. 9.5. Lastly, related work is highlighted in Sec. 9.6.

9.1 Fully-Streaming Neighbor Search Algorithm

We introduce our neighbor search algorithm and explain how it fundamentally improves the DRAM access efficiency by allowing completely streaming memory accesses (Sec. 9.1.1). We then describe the co-designed neighbor search hardware (Sec. 9.1.2). Finally, we discuss the key knob in our algorithm that dictates the accuracy-vs-performance trade-off (Sec. 9.1.3).

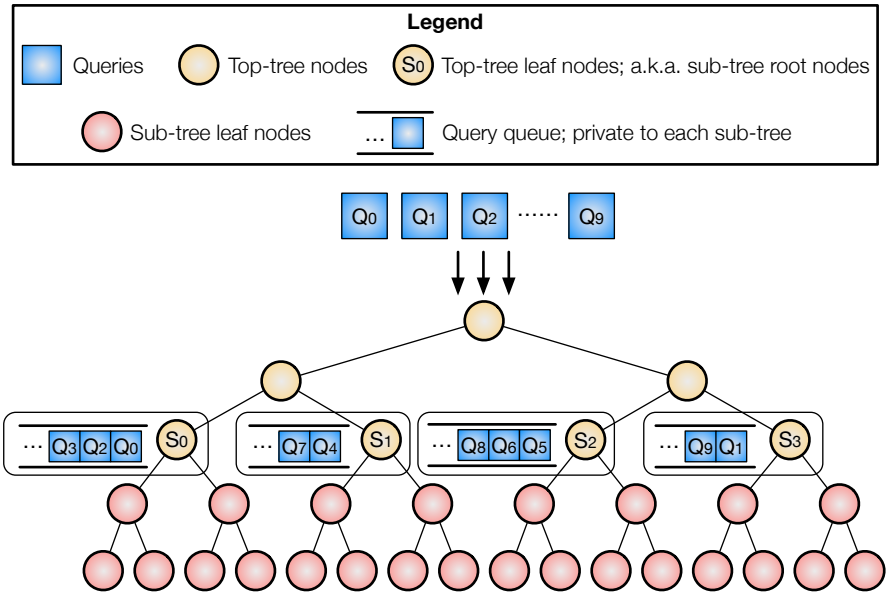


Fig. 9.1: The two-level tree data structure of our neighbor search algorithm. In the first stage, queries traverse the top-tree and are assigned to a particular sub-tree in the end. In the second stage, queries search neighbors in their assigned sub-tree, and backtracking is limited to within the sub-tree.

9.1.1 Algorithm

Our algorithm splits the K-d tree into a top tree and a set of sub-trees. Each top-tree leaf node is also the root node of a sub-tree. The search is then naturally divided into two stages: a top-tree search stage and a sub-tree search stage. The two stages themselves are massively parallel but are serialized with each other. Fig. 9.1 illustrates the idea.

In the first stage, all the queries search the top-tree (a binary search tree) until they reach the leaf nodes of the top-tree, at which point the queries are assigned to the corresponding sub-trees. Conceptually, each sub-tree has a queue that stores all the incoming queries. At the end of the first stage, queries in the sub-tree queues are written back to the memory in preparation for the second stage. In actual hardware, a queue has a fixed size. Thus, the memory write-back is phased, as we will discuss later.

Once all the queries finish the first stage, the algorithm enters the second stage,

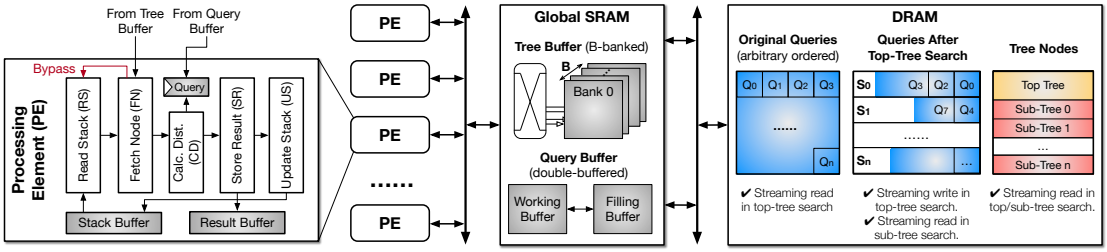


Fig. 9.2: Neighbor search hardware engine, which enables fully-streaming access to DRAM. The same hardware is used for both the top-tree search and the sub-tree searches, simplifying the hardware design.

where queries in each sub-tree are searched against the corresponding tree. For each sub-tree, the search process is exactly the same as that in the top-tree with a critical difference: queries are allowed to backtrack when they reach a leaf node of the sub-tree. This is necessary for a query to find all its neighbors.

However, we limit the backtracking to the sub-tree. The intuition is that nodes in other sub-trees are naturally far away from the query and thus are less likely to be neighbors. Architecturally, this ensures that each sub-tree and each query is loaded to SRAM once — at a cost of accuracy loss. We will discuss the accuracy implication of this design decision in Sec. 9.1.3 and how to mitigate the accuracy loss through approximation-aware network training in Sec. 9.3.

9.1.2 Hardware Design

The hardware designed to exploit the algorithm is shown in Fig. 9.2. The search is carried out by a set of PEs, each of which can execute a query independently. The PEs access data from the on-chip SRAM that stores various data structures. The SRAM interfaces with the DRAM through a DMA, as all DRAM accesses are streaming.

SRAM The SRAM is split into two global buffers and two local buffers. The global tree buffer and query buffer are accessed by all the PEs. Each PE is also equipped with a local result buffer and a local stack buffer private to each query.

The global tree buffer is accessed by the PEs simultaneously. To sustain a high read bandwidth, the tree buffer is heavily banked. Unlike in regular kernels, bank conflicts here could not be avoided by optimizing the data layout in the banks, because the access pattern of the PEs is known only at run time. We will show in Sec. 9.2 how to mitigate the performance impact of bank conflicts.

PE Design The PE design follows the algorithm of how a query traverses the K-d tree to search for its neighbors. As shown in the left blown-up panel in Fig. 9.2, a PE is pipelined into five stages, starting from reading the top of the traversal stack (RS) to fetch the next tree node to visit (FN), followed by calculating the distance between the query and the tree node (CD), storing results (SR) when a neighbor is found, and ended with updating the stack (US). The pipeline stalls only when the FN stage meets a bank conflict when reading the global tree buffer.

Hardware Reuse Due to the uniform traversal-based search in both top- and sub-tree searches, the hardware is reused in both phases. For instance, the PEs are designed with the generic traversal logic that is agnostic to what the search tree is and what the queries are. The US stage is skipped/bypassed in the top-tree search where no backtracking takes place (i.e., no update to the query stack).

The SRAM is also reused between the two phases. Specifically, the PE-local result buffer is re-purposed between storing the sub-tree queues in the top-tree search phase and storing the neighbor results in the sub-tree search phase. The global tree buffer is re-purposed between storing the top-tree and storing the sub-tree. During top-tree search, whenever a result buffer is full all the queries assigned to that queue (thus far) are streamed back to the DRAM.

9.1.3 Accuracy and Performance Trade-off

A key parameter that governs our algorithm is the top-tree height (TTH). TTH must be set to ensure both the top-tree and the sub-trees can be held in the on-chip SRAM. At

the same time, TTH also dictates the performance-vs-accuracy trade-off. We explore the implication of TTH in this section.

First, the top-tree height is dictated by the tree buffer size. We require that the entirety of the top-tree or a sub-tree, while is being searched, is completely stored in the tree buffer. This ensures that the PE pipeline does not stall because the required data are off-chip. Thus, the top-tree height h_t must be within the range $[\mathcal{H} + 1 - \log_2(\mathcal{S} + 1),\ \log_2(\mathcal{S}+1)]$ to satisfy the following two inequalities, where $\mathcal{H}$ is the total K-d tree height and $\mathcal{S}$ is the total tree buffer size:

$$2^{h_t} - 1 \leq \mathcal{S} \tag{9.1}$$

$$2^{\mathcal{H}-h_t+1} - 1 \leq \mathcal{S} \tag{9.2}$$

Given that a TTH is within the permissible range, a shorter top-tree increases the neighbor search accuracy at a cost of more computation, and vice versa. This can be explained by a first-order analytical model, where the total number of nodes a query accesses is proportional to the sum of:

i the number of tree nodes that are visited during the forward traversal, i.e., from the root node of the top-tree to a leaf node of a sub-tree,

ii the number of nodes that are visited during the sub-tree backtracking.

The cost of (i) is constant, as it depends only on the total tree height. The cost of (ii) inversely depends on the TTH: a taller top-tree translates to visiting fewer nodes in the sub-tree backtracking, reducing the cost of (ii) and, by extension, the total cost. Fig. 9.3 quantifies how the total number of nodes accessed per query (y-axis) varies with the TTH (x-axis) using the average statistics of PointNet++(c) on the KITTI dataset. As the TTH increases to 10, only 2% of the tree nodes are accessed by a query. Visiting fewer nodes improves the search speed but also degrades the accuracy.

An assumption we make, as with Tigris (Xu et al., 2019) and QuickNN (Pinkham et al., 2020b), is that a sub-tree can be stored completely on-chip. This is a reasonable

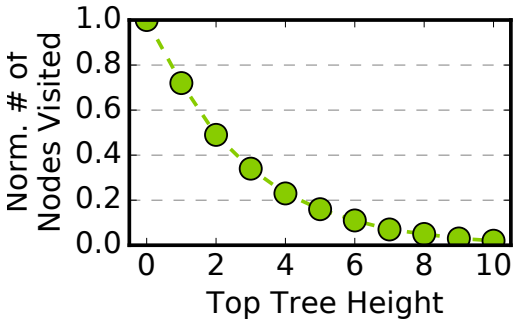


Fig. 9.3: Number of tree nodes visited per query reduces as the top-tree height increases.

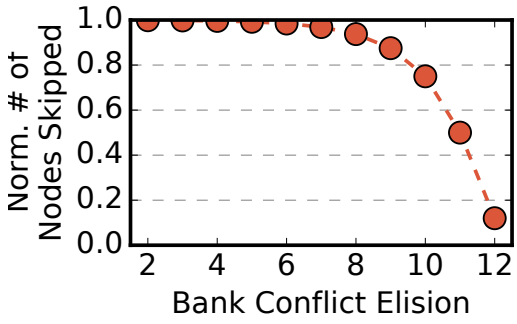


Fig. 9.4: Number of tree nodes skipped per query reduces as the elision height increases.

assumption: a typical 10 MB point cloud using a 5-level top-tree would result in a sub-tree size of 640 KB, smaller than a typical on-chip buffer size found in mobile SoCs. In case of excessively large point clouds, CRESCENT can in theory recursively split a sub-tree; we do not observe this need in common datasets.

9.1.4 Efficiency Discussion

The split-tree algorithm enables completely streaming DRAM accesses. The panel on the right of Fig. 9.2 shows how the different data structures are laid out in the DRAM and how they are accessed in a streaming fashion. Converting random DRAM accesses to streaming accesses reduces the DRAM energy (Micron, 2020; Gao et al., 2017), and enables double-buffering, which improves performance because: 1) off-chip data accesses are overlapped with computation, and 2) data needed by the datapath are readily available on-chip without stall.

Compared to prior neighbor search algorithms that also enable streaming accesses such as Tigris (Xu et al., 2019) and QuickNN (Pinkham et al., 2020b), we reduce both the search load and DRAM traffic. We qualitatively discuss it here, and quantify the gains in Sec. 9.5.5.

First, Tigris and QuickNN use exhaustive search within the sub-trees, whereas we retain K-d tree search in the sub-trees, thereby reducing the total search load. Retaining K-d tree search in the sub-trees is not an obvious design decision, because it introduces irregular *on-chip* memory accesses in the form of bank conflicts, which prior work aims to avoid at a cost of more search work.

Our strategy is different: we reduce the search work by retaining K-d tree search and mitigate the resulting irregular on-chip memory accesses through inference-training co-design. Specifically, we will show a selective bank conflict elision scheme to significantly reduce bank conflicts (Sec. 9.2), which, when coupled with an approximation-aware training procedure (Sec. 9.3), retains the application accuracy.

Second, we reduce the amount of DRAM accesses compared to Tigris and QuickNN, both of which load (and reload) a sub-tree from DRAM whenever the corresponding query buffer is full. We instead first stage all the queries to a sub-tree in DRAM and then process them in a batch, thus loading each sub-tree exactly once.

9.2 Selective Bank Conflict Elision

This section addresses inefficiencies pertaining to on-chip memory accesses. We first describe our main idea of selectively eliding bank conflicts (Sec. 9.2.1). We then discuss how point cloud algorithms proceed when bank conflicts are elided (Sec. 9.2.2) and the hardware support (Sec. 9.2.3). Finally, we identify the key knobs that dictate the accuracy-vs-performance trade-off (Sec. 9.2.4).

9.2.1 Main Idea

A key requirement of the SRAM design is to feed data required by the PEs without stalling them. Such a requirement is easy to meet in conventional DNNs or other regular kernels, where data access patterns are statically known and thus SRAM data layout can

be statically optimized accordingly (Zhou et al., 2021). The on-chip memory access patterns in point cloud algorithms, however, are only dynamically known, introducing SRAM bank conflicts that are detrimental to overall performance.

Motivated by the error-tolerance nature of neural networks, our idea is to dynamically ignore bank conflicts when appropriate. That is, when multiple memory requests fall in the same bank, instead of serializing the accesses we allow only one request to access the SRAM; other requests return immediately without stalling. While conceptually simple, actually realizing this idea requires answering three questions:

1. What data should conflicted accesses return, and how should the algorithm proceed without the correct data?
2. How to support bank conflict elision in hardware?
3. When is it appropriate to elide bank conflicts without accuracy drop?

The answers to these questions depend on where a bank conflict takes place in the algorithm, because different memory accesses request data of different significance. Both neighbor search stage and feature computation introduce bank conflicts. In neighbor search, bank conflicts are caused by accessing the tree buffer; all other accesses are regular. In feature computation, aggregating neighbors of a point as inputs to the MLP causes bank conflicts; SRAM accesses incurred during MLP are regular. We now elaborate on how the three questions above are addressed in both stages.

9.2.2 How Algorithms Proceed with Bank Conflicts Elision

Feature Computation To aggregate neighbors, SRAM accesses are made to retrieve neighbors of a point. Thus, ignoring a conflicted access essentially ignores a point's neighbor, in which case we must fill in the missing neighbors, as the subsequent MLP anticipates an input matrix of a given size (decided at the training time).

To meet the size requirement, we propose to simply reuse the point returned from the request that *is* allowed to access the bank. The intuition is that concurrent accesses, say A and B, are guaranteed to be requesting neighbors of the same point P (Feng et al., 2020b). Reusing the returned data from A for B is equivalent to replicating one of P's neighbors. This replication strategy is commonly done in point cloud network design to meet the size requirement in case a neighbor search does not return enough neighbors (Qi et al., 2017a,b). Our design essentially performs this replication in hardware, implicitly.

Neighbor Search The situation is slightly different for neighbor search, where bank conflicts happen when the PEs access the tree nodes during tree traversal. One could use the same replication strategy used in the feature computation stage: if accesses A_1 from PE 1 and A_2 from PE 2 conflict on the same bank, reuse the data returned from A_1 for A_2. However, this could lead to side effects such as program crash, redundant computation, and infinite loop. For instance, when the node returned from A_1 is in the part of the tree that PE 2 has already visited, pushing A_1 onto PE 2's stack leads to an infinite loop or, at least, redundant traversals.

Our design simply ignores the conflicted accesses. Upon a conflict, the FN stage in a PE skips the remaining pipeline stages and reads the next item on the stack. This is denoted by the "bypass" signal in the PE shown in Fig. 9.2. Algorithmically, this is equivalent to skipping all the nodes beneath the lost node during tree traversal. This strategy omits potential neighbors but guarantees that the traversal terminates.

An optimization that we leave for future work is to check whether the node returned from A_1, the request allowed to access the SRAM, is beneath the node (in the tree) that would have been returned from A_2 if the bank conflict were to be observed; if so, using A_1 to continue the search in P_2 is guaranteed to terminate without side effects. Doing so would skip fewer nodes and potentially increase the accuracy.

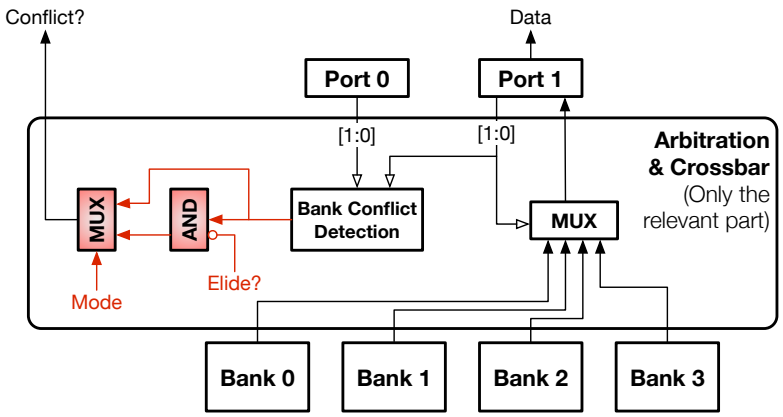


Fig. 9.5: Supporting bank conflict elision is trivial in hardware, as many existing hardware structures can be reused. The shaded/colored components are the augmentation, which is required for each SRAM port. Only the relevant part of the hardware is shown for simplicity. The *Mode* signal selects between the neighbor search mode and the feature computation mode. The `AND` gate lowers the *Conflict* signal when bank conflict elision is enabled in the neighbor search stage.

9.2.3 Hardware Support

Eliding bank conflicts is virtually free to implement in hardware by using many existing structures in banked SRAM design. As an example, Fig. 9.5 shows a simple banked SRAM with 2 ports and 4 banks. The key to a banked SRAM is the arbitration and crossbar logic, which detects bank conflicts and routes data from a bank to the right port (a MUX here). For simplicity, we show only the relevant hardware and assume a low-order interleaving, i.e., the two least significant bits in the address select a bank.

Assume both accesses from the two ports fall into Bank 0, and Port 0 is allowed access. In the baseline SRAM, the MUX before Port 1 would select data returned from Bank 0, but this data will be ignored because the bank conflict detection logic would raise the *Conflict* signal, indicating to Port 1 that a bank conflict occurs and the memory request is to be issued again. But, critically, the data returned from Bank 0 is exactly what Port 1 needs in the feature computation stage under bank conflict elision. We

simply lower the *Conflict* signal in this case, which is accomplished by ANDing the output of bank conflict detection and the negation of the *Elide* signal, which indicates whether bank conflict elision is enabled.

The *Mode* signal operates a MUX to select between the neighbor search vs. feature computation mode. In neighbor search, the original bank conflict signal is used, except the PE will not re-issue the memory request; instead, the PE simply continues the search with the next item on the stack.

9.2.4 When to Elide Bank Conflicts?

Eliding bank conflicts returns incorrect data to the PEs and, thus, hurts accuracy. We find that eliding bank conflicts in feature computation leads to little to no accuracy loss whereas eliding bank conflicts during neighbor search, without care, has significant accuracy implications (Sec. 9.5.3). This is because in feature computation the data that would have been returned (if bank conflicts were observed) are replaced with the data returned from the conflicting access; in neighbor search, however, eliding bank conflicts directly skips all the computations associated with that node altogether. We thus focus on the neighbor search stage here.

Intuitively, the accuracy loss is smaller when ignoring a memory access made to a lower-level tree node, as fewer tree nodes would be skipped later in the traversal. Fig. 9.4 shows how the percentage of skipped tree nodes (y-axis) varies with the tree level below which bank conflicts are elided (x-axis). The data are averaged across all the queries of PointNet++(c) on the ModelNet dataset, where the total tree height is 14. When bank-conflicted accesses below level 2 are ignored, almost 100% of the tree nodes are skipped, which degrades the model accuracy to almost zero (not shown). When the elision level is 12, only 10% of the tree nodes are skipped.

Skipping more nodes degrades accuracy but increases the search speed. Therefore, a natural knob that controls the trade-off of accuracy-vs-performance is the *elision height*

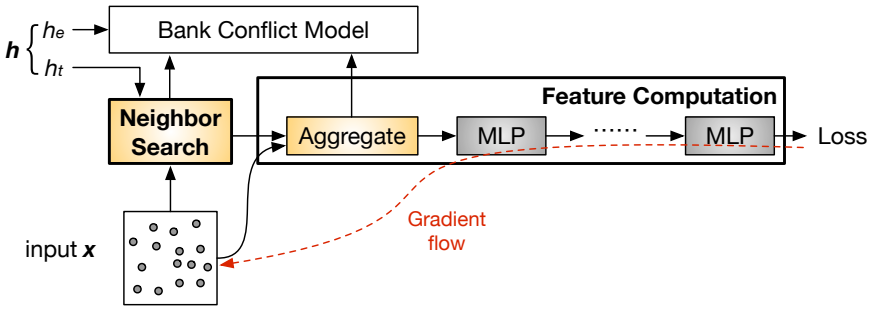


Fig. 9.6: Training a point cloud network with approximate neighbor search and bank conflict elision. Note that the training is end-to-end differentiable as in conventional DNN training. The non-differentiable parts, neighbor search and aggregation, do not participate in the gradient flow.

h_e, which is defined as the tree level beneath which all conflicted memory accesses are ignored. Sec. 9.3 will show how incorporating h_e into model training can minimize the accuracy loss while providing the accuracy-vs-performance trade-off without retraining.

9.3 Approximation-Aware Network Training

Our neighbor search algorithm and bank conflict elision, if applied directly on a trained point cloud DNN at inference time, will decrease the accuracy sharply (Sec. 9.5.1). This is because the original network is not trained with the various approximation techniques in mind. To mitigate the accuracy drop, we propose a modified network training procedure that mitigates the accuracy loss.

The goal here is to learn a DNN that retains a high accuracy under approximation compared to the baseline network. In particular, we consider two approximation knobs: the top-tree height h_t and the elision height h_e.A larger h_t decreases accuracy but increases the performance; conversely, a larger h_e increases the accuracy at a cost of lower performance.

A straightforward idea is to integrate $\mathbf{h} = \langle h_t, h_e \rangle$ as part of the inference such that the DNN is trained for a particular $\mathbf{h}$. In essence, this is similar to fine-tuning a compressed model to regain the accuracy, where a network learns to adjust its weights given the approximation introduced by a particular compression setting.

While one could train a dedicated model for each possible $\mathbf{h}$ and build an ensemble, that would increase the training overhead and deployment complexity. Instead, we propose to learn one model that adapts to different $\mathbf{h}$. Mathematically, we aim to learn a DNN distribution $f(\cdot, \mathbf{h}; \theta) \sim F$ such that different DNNs sampled from the distribution F share the same model parameter θ and provide similar accuracy given an input $\mathbf{h}$ (along with the input point cloud).

To that end, our training procedure augments the conventional training with one simple extension: conventional training samples input data; our training also randomly samples an $\mathbf{h}$ *for each input*. During the forward propagation, $\mathbf{h}$ is used to modulate the neighbor search and bank conflict elision. In this way, the model parameter θ is trained to accommodate the approximations introduced during the forward inference. The training flow is shown in Fig. 9.6.

In order to replay the same inference-time approximation during training, we integrate a hardware simulator for modeling the bank conflict. The bank conflict model is called by both neighbor search and feature computation (the aggregation operation), as Fig. 9.6 shows. The bank conflict simulator takes in two parameters: 1) h_e, which indicates the tree level below which bank conflicts are elided, and 2) the hardware banking configuration (e.g., number of banks, bank size). We find that training with the exact banking configuration on the inference hardware yields higher accuracy, but absent an exact hardware configuration training with a generic banking configuration provides noticeable benefits, too (Sec. 9.5.3).

Finally, note that neighbor search and aggregation do not participate in gradient descent; they simply construct inputs to the MLP layers. Thus, the model is end-to-end differentiable even though neighbor search and aggregation are not.

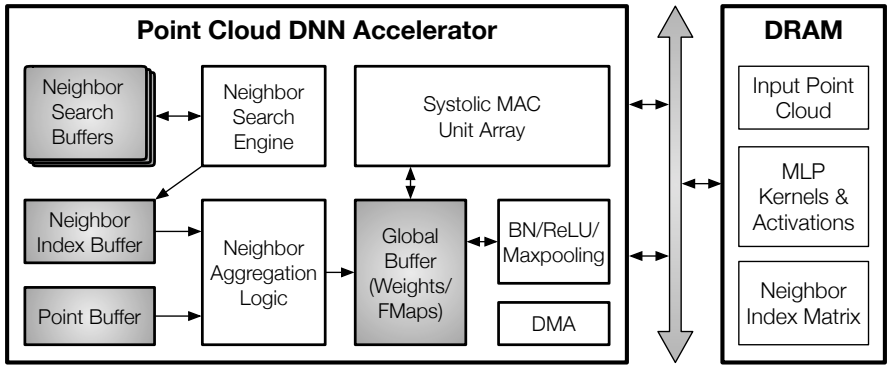


Fig. 9.7: Overall architecture of the point cloud DNN accelerator, which includes three main components: a Neighbor Search Engine, an Aggregation Unit, and a systolic array for executing the MLPs in feature computation. The Neighbor Search Buffers include all the buffers shown in Fig. 9.2.

9.4 Experimental Setup

Architecture Design Fig. 9.7 shows the overall point cloud accelerator, which includes three main components: a neighbor search engine as described in Sec. 9.1.2, a neighbor aggregation unit, which uses the design in Mesorasi (Feng et al., 2020b), and a DNN accelerator for executing the MLPs. Without losing generality, we assume a systolic-array-based DNN accelerator, which is configured to have a 16×16 MAC array, where each MAC unit mimics the design of that in the TPU (Jouppi et al., 2017a).

The on-chip SRAM is partitioned to serve different purposes. The global buffer serves the weight and activations for the systolic array. It is configured to be 1.5 MB in size. The Point Buffer is a 64 KB 16-banked buffer serving points during aggregation. The Neighbor Index Buffer is sized at 12 KB with a single bank. The Tree buffer and the Query buffer are sized at 6 KB and 3 KB with 4 banks and 1 bank, respectively. These two buffers support selective bank elision as described in Sec. 9.2.3. The neighbor search engine has 4 PEs, each with a dedicated result buffer and a stack buffer, which are sized at 1.5 KB and 256 B, respectively.

Experimental Methodology We synthesize, place, and route the datapath of the

Table 9.1: Evaluation models.

Application Domains	Algorithm	Dataset
Classification	PointNet++ (c) DensePoint	ModelNet40
Segmentation	PointNet++ (s)	ShapeNet
Detection	F-PointNet	KITTI

neighbor search engine, the systolic array, and the aggregation unit using an EDA flow consisting of Synopsys and Cadence tools with the TSMC 16 nm FinFET technology. The SRAMs are generated using the Arm Artisan memory compiler. Power is estimated using Synopsys PrimeTimePX by annotating the switching activity. We then build a cycle-accurate simulator of the architecture with the latency of each component parameterized from the post-synthesis results of the RTL design.

The DRAM is modeled after Micron 16 Gb LPDDR3-1600 (4 channels) according to its datasheet (Micron, 2014). The DRAM energy is obtained using Micron System Power Calculators (Micron, 2020). On average, the energy ratio between a random DRAM access and a streaming DRAM access is about 3:1, and the energy ratio between a random DRAM access and an SRAM access is about 25:1, both matching prior work (Gao et al., 2017; Yazdanbakhsh et al., 2018).

Software Setup Tbl. 9.1 lists the four point cloud networks used in the evaluation, which covers common point cloud tasks including classification, segmentation, and detection. For classification, we evaluate the classic PointNet++(c) (Qi et al., 2017b) and DensePoint (Liu et al., 2019b) on the ModelNet40 dataset (Wu et al., 2015). We use the overall accuracy as accuracy metric. For segmentation, we evaluate PointNet++(s) (Qi et al., 2017b) on the ShapeNet dataset (Chang et al., 2015). The metric used in segmentation is the standard Intersection-over-Unit (mIoU) accuracy. For detection, we evaluate F-PointNet (Qi et al., 2018) on the KITTI dataset (Geiger et al., 2012a) and

report the geometric mean of the IoU metric on the car class.

To ensure that the improvements from CRESCENT are not due to the inefficiencies of the network implementation, we use the versions of these models optimized by Feng et al. (Feng et al., 2020b), which removes redundant MLP computations and on average achieves 1.6× speedup over the corresponding author-released implementations.

Baseline We compare against three baselines:

- GPU: the mobile Pascal GPU on Nvidia's Jetson TX2 development board (Nvidia, 2017b).
- TIGRIS+GPU: this baseline executes the neighbor search on Tigris (Xu et al., 2019), a recent neighbor search accelerator that does not perform approximate neighbor search and selectively bank conflict elision, and executes the feature computation on the mobile Pascal GPU.
- MESORASI, a prior point cloud network accelerator (Feng et al., 2020b) that uses Tigris (Xu et al., 2019) for neighbor search and executes the feature computation on a dedicated systolic-array without selectively bank conflict elision. The exact same systolic array configuration is used in CRESCENT with the exception that CRESCENT performs selective bank conflict elision.

Area Overhead Our accelerator has a total area of $1.55\,\mathrm{mm}^2$, in which the CRESCENT-specific portion is almost negligible. The only hardware extension is one that selectively elides the bank conflict (Fig. 9.5), which requires an additional MUX and an `AND` gate for each port of the SRAM.

Traininig Overhead Our approximation-aware training increases the training time by 38%. The main overhead is to simulate bank conflicts, which currently is a multi-threaded CPU implementation. Using a random $\mathbf{h}$ does not further increase the training overhead, since we still perform one search per inference. Note that the training overhead is amortized across all subsequent inferences.

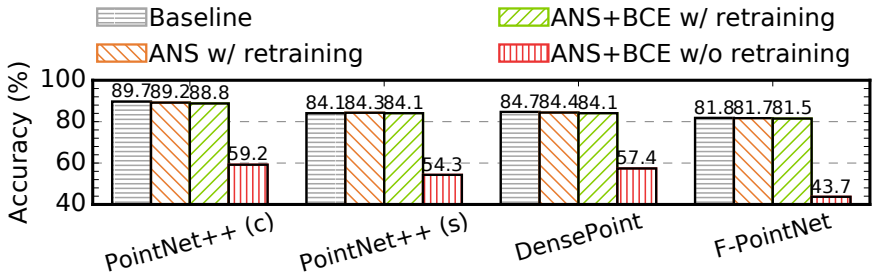


Fig. 9.8: Accuracy comparison between the baseline models, ANS+BCE without re-training, ANS with re-training under $h_t = 4$, and ANS+BCE with re-training under $h_t = 4$ and $h_e = 12$.

Variants We evaluate two variants of CRESCENT to decouple the contribution of the two optimizations:

- ANS performs approximate neighbor search but does not elide bank conflicts.
- ANS+BCE performs approximate neighbor search while also eliding bank conflicts in neighbor search and aggregation.

9.5 Evaluation

We first show that CRESCENT achieves similar accuracy as the baseline (Sec. 9.5.1) but delivers significant speedups and energy reductions (Sec. 9.5.2). We then provide a detailed analysis of our training procedure and understand how its effectiveness varies with respect to different algorithmic and hardware configurations (Sec. 9.5.3). We perform a sensitivity study to understand CRESCENT's performance and energy savings vary under different settings (Sec. 9.5.4). Finally, we provide a quantitative comparison with prior neighbor search accelerators (Sec. 9.5.5).

9.5.1 Accuracy

We find that directly applying CRESCENT optimizations without retraining significantly degrades the model accuracy. Integrating approximation into the training process elevates the accuracy to the baseline level. Fig. 9.8 compares the model accuracy between four schemes: 1) the baseline models, 2) ANS+BCE without re-training, 3) ANS+BCE with re-training, and 4) ANS with re-training. In this specific case, each re-trained model is trained specifically for the approximate setting where $h_t = 4$ and/or $h_e = 12$.

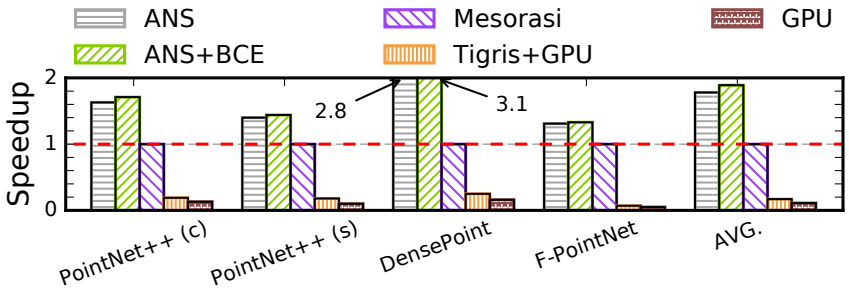
Directly applying the two optimizations at inference time degrades the accuracy between 27.3% to 40.5%, making the models practically useless. Re-training regains the accuracy with an accuracy drop of at most 0.9% (PointNet++(c)). In PointNet++(s), re-training completely recovers the accuracy loss introduced in approximation. The fact that we can almost completely recover the accuracy loss with ANS+BCE, the most aggressive approximation setting, shows the effectiveness of our approximation-aware training. The accuracy of ANS alone is slightly higher than that of ANS+BCE, as the latter applies two approximations whereas the former applies only one.

9.5.2 Performance and Energy

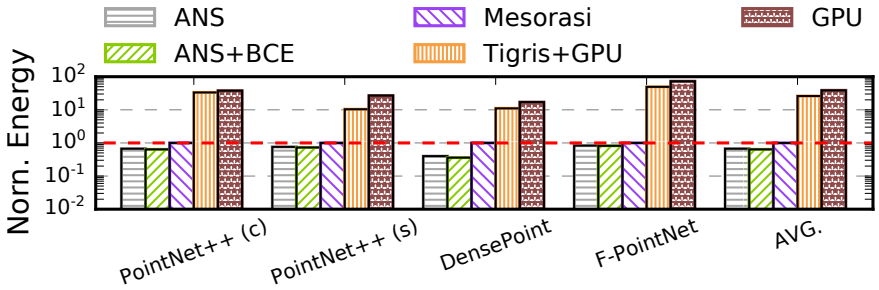
Using the re-trained ANS and ANS+BCE model shown in Fig. 9.8, we compare CRESCENT's performance and energy consumption over the baseline accelerator, shown in Fig. 9.9.

Speedup Fig. 9.9a shows the speedup of ANS and ANS+BCE against the three baselines; all data are normalized to MESORASI. Among the three baselines, TIGRIS+GPU and GPU are much slower than MESORASI, because the latter accelerates feature computation on a systolic array.

Overall, ANS and ANS+BCE achieve a $1.7\times$ and $1.9\times$ speedup, respectively, over MESORASI. Comparing the speed of ANS+BCE and ANS shows that approxi-



(a) Speedup. Higher is better.



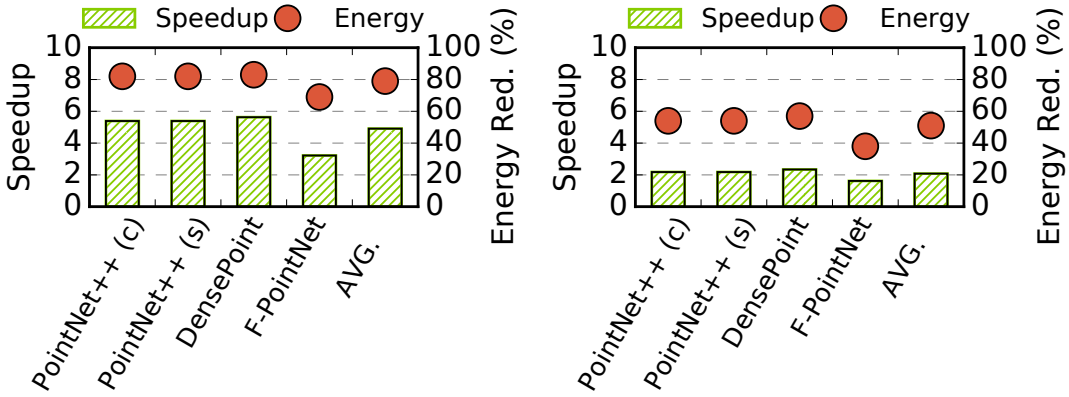
(b) Normalized energy. Lower is better.

Fig. 9.9: End-to-end speedup and normalized energy of ANS and ANS+BCE over the baseline.

mation neighbor search contributes more to the speedup than bank conflict elision. The speedups on DensePoint are the highest ($2.8\times$ and $3.1\times$, respectively) because DensePoint's time is dominated by neighbor search (81%) whereas neighbor search takes "only" about 55% of the time in other models.

To understand the sources of speedup, Fig. 9.10a and Fig. 9.10b show the speedup of ANS+BCE on neighbor search and on the aggregation operation in feature computation, respectively. On average, ANS+BCE achieves a $4.9\times$ speedup on neighbor search and a $2.1\times$ speedup on aggregation.

Energy Savings Fig. 9.9b shows the energy consumption of ANS and ANS+BCE normalized to MESORASI. On average, ANS and ANS+BCE save 33% and 36% of the total energy, respectively. The energy saving is mainly contributed by approximate neighbor search rather than bank conflict elision, because the former optimizes the



(a) Neighbor search.

(b) Aggregation.

Fig. 9.10: Speedup and energy savings of ANS+BCE on neighbor search and aggregation alone.

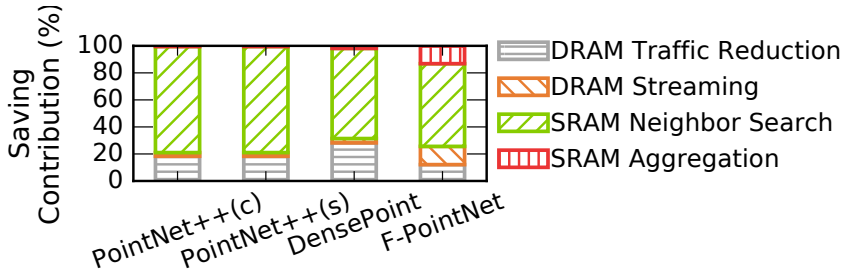


Fig. 9.11: Memory energy saving contribution.

DRAM traffic, which contributes more to the energy than the SRAM traffic, which the latter optimizes for. DensePoint, again, has the highest energy saving because it is dominated by neighbor search. As a comparison, TIGRIS+GPU and GPU consume $25\times$ and $38\times$ more energy, respectively, compared to MESORASI.

Fig. 9.10a and Fig. 9.10b on the right y-axes show the energy savings on neighbor search and aggregation. DensePoint's savings on these two operations *in isolation* are on par with other networks, confirming that its significant end-to-end savings are primarily attributed to the dominance of neighbor search in its execution time.

Tease Apart Contributions To understand the sources of energy savings, Fig. 9.11 decouples the memory energy savings into four components: converting random

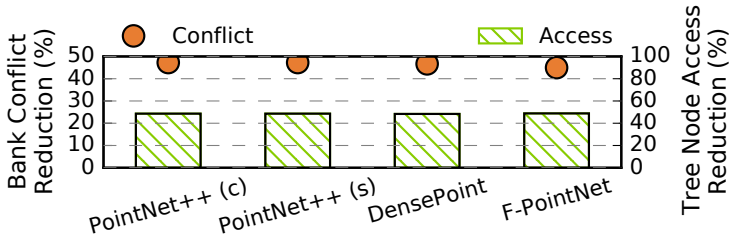


Fig. 9.12: Tree node access saving and bank conflict reduction of ANS+BCE.

DRAM accesses to streaming accesses, DRAM traffic reduction, SRAM traffic reduction in neighbor search, and SRAM traffic reduction from aggregation. The former two are due to the new neighbor search algorithm, and the latter two are due to the selective bank conflict elision optimization.

The main energy saving contributor is the SRAM traffic reduction in neighbor search, which frequently accesses the Tree Buffer. While the DRAM savings are relatively smaller, we expect the DRAM savings will become more significant in the future as the point clouds grow in size.

We quantify the impact of selective bank conflict elision (BCE) in Fig. 9.12, where we show the reduction in bank conflicts (left y-axis) and, as a result, the reduction in the number of tree nodes visited (right y-axis). The results are obtained by comparing ANS+BCE with ANS. Overall, BCE avoids over 45% of bank conflicts and reduces 50% of tree node accesses in neighbor search. This result explains the $1.9\times$ speedup over MESORASI by ANS+BCE.

9.5.3 Understanding the Training Procedure

We use PointNet++(c) as a representative model to drive the analyses in this section. The conclusions generally hold.

Dedicated Models We first evaluate the accuracy of models trained with dedicated approximation settings.

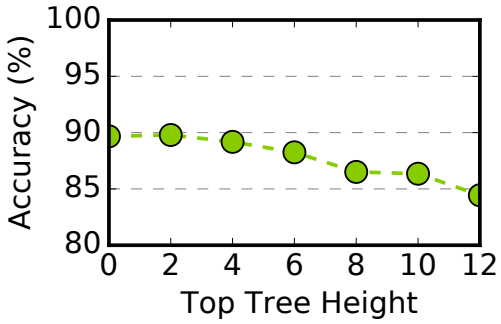


Fig. 9.13: Accuracy of dedicated PointNet++(c) models under different top-tree heights (h_t).

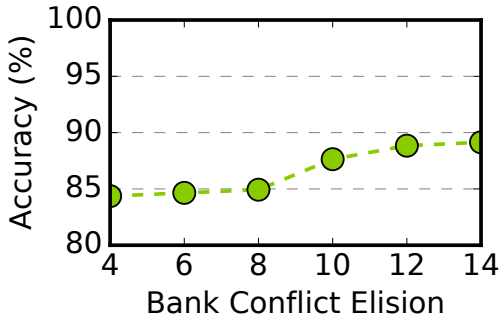


Fig. 9.14: Accuracy of dedicated PointNet++(c) models under different elision heights (h_e).

Fig. 9.13 shows the accuracy of PointNet++(c) trained under different top-tree heights (h_t) and then inferenced under the same h_t. The setting h_t being 0 is the baseline model with exact search. As the h_t increases, the accuracy decreases. This is because a larger h_t reduces the search space and, thus, it is less likely to find the exact neighbors for each query. The accuracy is acceptable initially, dropping from 89.6% to 88.8% as h_t increases from 0 to 4. Beyond 4, the accuracy drop becomes more significant. As the top-tree height reaches 12, the accuracy is only 84.4%. As we will show later, however, a higher h_t leads to a higher speedup, providing a large trade-off space.

Fig. 9.14 performs a similar study while varying the elision height h_e. Each marker in the figure represents a dedicated ANS+BCE model trained with different h_e ranging from 4 to 14; h_t in this example is fixed at 4. As h_e increases, the accuracy increases. This is because a higher elision height skips fewer tree nodes during tree traversal, leading to a better search result. At a h_e of 12, the accuracy loss is only 0.8%. The accuracy loss is over 5% when h_e reduces to 4, essentially ignoring almost all nodes in the sub-tree.

Mixed Training We now evaluate how a model trained by sampling approximation settings adapts to different approximation levels at inference time. Fig. 9.15 compares three schemes: 1) a model trained with $h_t = 1$, 2) a model trained with $h_e = 6$, and 3)

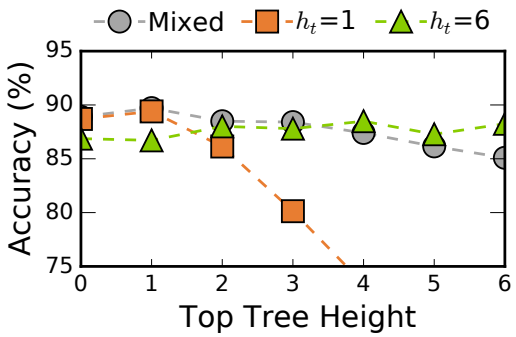


Fig. 9.15: Accuracy comparison of different training schemes.

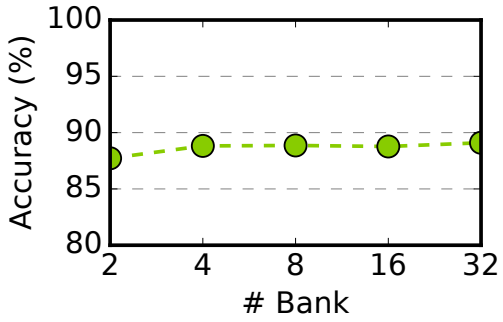


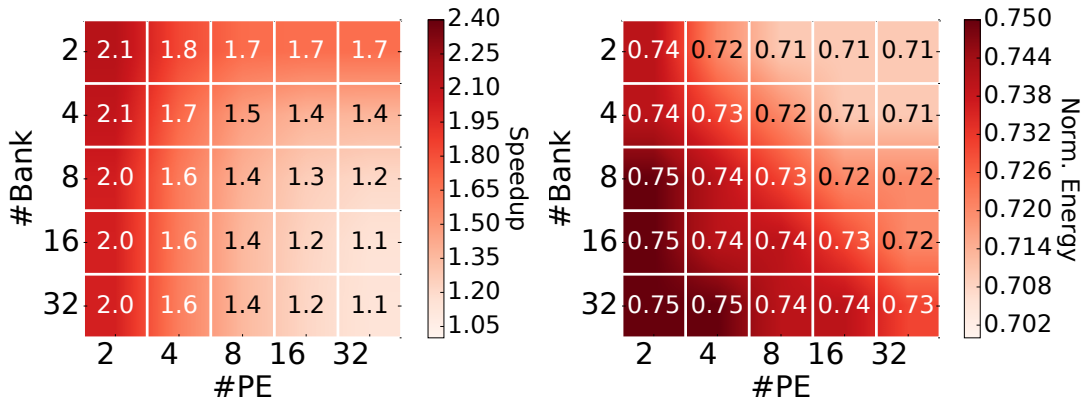
Fig. 9.16: Sensitivity of training accuracy to bank conflict configuration.

a model trained by random sampling h_t between 1 and 6 for each input ("Mixed" in the figure). We show their accuracy under different inference-time h_t.

When a dedicated model is trained with $h_t = 1$, the accuracy significantly drops when the inference-time h_t is greater than 1. This is not surprising: a model trained with little approximation in mind does not perform well when inference performs aggressive approximation. When a dedicated model is trained with $h_t = 6$, however, it performs reasonably well across different h_t at inference time, even for h_t settings that are not seen in the training time.

The mixed model consistently provides higher or similar accuracy compared to the dedicated $h_t = 1$ model. Compared to the dedicated $h_t = 6$ model, the mixed model is significantly better when higher accuracy is required (i.e., $h_t \leq 3$). The accuracy is only noticeably worse than the dedicated $h_t = 6$ model when the inference-time h_t is 6, which is what the dedicated $h_t = 6$ model is trained to do well on. The mixed model is favorable when the accuracy requirement is high, which is arguably more important than the low-accuracy regime.

Bank Conflict Simulation In order to integrate bank conflict elision into training, we simulate the bank conflicts in the forward propagation process during training. However, at training time the exact banking configuration of the target hardware might be unknown. Fig. 9.16 show the accuracy of training a model assuming 4 banks in the



(a) Speedup sensitivity.

(b) Energy sensitivity.

Fig. 9.17: Sensitivity of speedup and (normalized) energy to hardware configuration (PE and bank counts) on PointNet++(c).

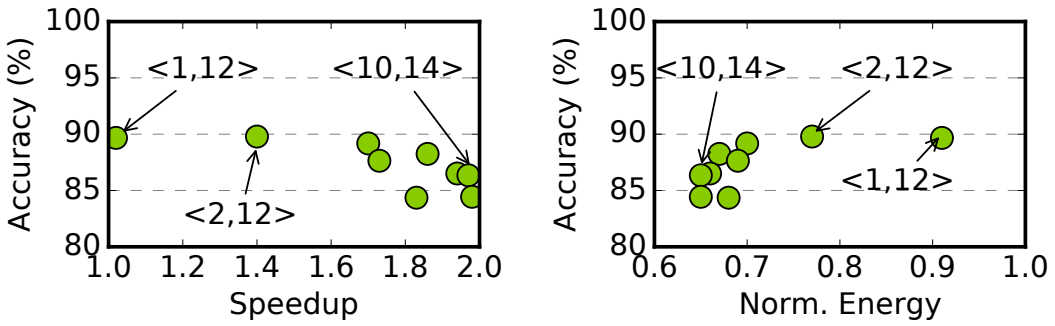
SRAM while inferencing under different numbers of banks. The accuracy beyond 8 is largely stable; the accuracy has about 2% drop when inferencing on a 2-banked SRAM.

BCE in Aggregation vs. Neighbor Search We perform bank conflict elision in both neighbor search and feature aggregation. We find that the overall accuracy is insensitive to bank conflict elision in aggregation even *without* re-training. Across all networks, applying bank conflict elision in aggregation alone (while turning off other approximations) results in at most 0.3% accuracy loss. In contrast, accuracy typically drops by double digits if bank conflict elision is applied in neighbor search without re-training. As discussed in Sec. 9.2.4, this is because in the latter case eliding bank conflicts completely skips subsequent search operations.

9.5.4 Sensitivity Study

Hardware Configuration Fig. 9.17a and Fig. 9.17b show how CRESCENT's speedup and energy vary, respectively, as the numbers of PEs and the number of Tree Buffer banks vary. The energy is normalized to the corresponding baseline.

Naturally, the speedup is higher on less-capable baselines and diminishes on more



(a) Speedup-vs-accuracy.

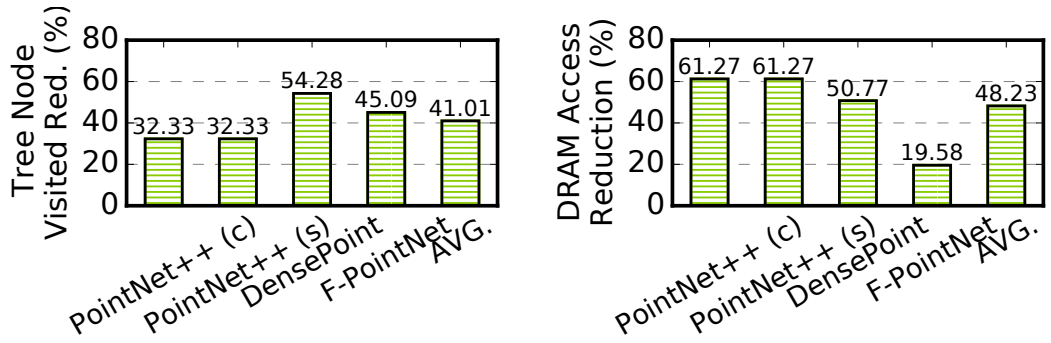
(b) Energy-vs-accuracy.

Fig. 9.18: Accuracy vs. performance vs. energy trade-off on PointNet++(c) under different $\langle h_t, h_e \rangle$ combinations.

capable baselines (e.g. 32 PEs and 32 banks), because performance optimizations are less important when the hardware is faster to begin with. Note, however, that a 16-bank memory introduces a large cross-bar overhead and is generally impractical for mobile-grade accelerators (Agarwal et al., 2009; Zhou et al., 2021).

The significant energy saving is consistent across hardware configurations. Even with a 32 PE 32 bank configuration, CRESCENT still saves about 27% of energy on PointNet++(c). This is because the energy is roughly proportional to the amount of work done. Changing the hardware configuration does not affect the bulk of the work needed to be done.

Approximation Degrees Fig. 9.18a and Fig. 9.18b show the accuracy-vs-speedup and accuracy-vs-energy trade-offs, respectively, with different h_t and h_e combinations, which dictate different approximation strengths. The data are reported from PointNet++(c), but the trend generally holds. Overall, varying h_t from 0 to 12 and h_e from 4 to 14 provide a trade-off space of about 5% accuracy range, 2.0 $\times$ performance range, and 1.5 $\times$ energy range.



(a) Reduction in tree nodes visited from Tigris (Xu et al., 2019).

(b) DRAM access (in Bytes) reduction from QuickNN (Pinkham et al., 2020b).

Fig. 9.19: Comparison with prior neighbor search accelerators Tigris and QuickNN.

9.5.5 Comparison with Prior Neighbor Search Accelerators

QuickNN (Pinkham et al., 2020b) and Tigris (Xu et al., 2019) are two recent neighbor search accelerators using a split-tree data structure. As discussed in Sec. 9.1.4, CRESCENT reduces both the search load and DRAM traffic. Fig. 9.19a shows that the K-d tree-based search reduces the total number of tree nodes visited by 41% compared to exhaustive search. This explains the one-order-of-magnitude performance improvement over the Tigris-based accelerator shown in Sec. 9.5.2.

QuickNN, similar to CRESCENT, also presents streaming DRAM accesses — at the expense of redundant DRAM accesses, since each sub-tree is potentially loaded onto the accelerator multiple times. Compared to a QuickNN implementation with the same PE configuration, Fig. 9.19b shows that CRESCENT reduces the total DRAM accesses by 48%.

Finally, we target DNN-based algorithms and, thus, can mitigate the potential accuracy loss through end-to-end network training, which is not available to QuickNN and Tigris; both target a non-DNN algorithm (point cloud registration).

9.6 Related Work

Deep Learning for Point Clouds Point cloud algorithms are increasingly moving toward DNNs, which has spurred recent interests in accelerating point cloud networks (Feng et al., 2020b; Lin et al., 2021; Hyun et al., 2021). Point cloud DNNs mainly come in two forms: one that operates on raw points (Qi et al., 2017a,b; Wang et al., 2019a; Zhang et al., 2019a; Liu et al., 2019b), and the other that first voxelizes points and operates on voxels, which are grid-aligned points (Graham et al., 2018; Choy et al., 2019). The former requires explicit neighbor search whereas the latter accesses neighbors through simple indexing. It is unclear whether future point cloud algorithms will definitively favor one form over the other. CRESCENT focuses on optimizing point-based algorithms, whose flexibility and compact data representations are shown to be critical in many application domains (Guo et al., 2020), such as object detection, localization (SLAM), segmentation, and classification.

PointAcc (Lin et al., 2021), Point-X (Zhang and Zhang, 2021), and Mesorasi (Feng et al., 2020b) are all recent point cloud accelerators. They are fundamentally orthogonal to our work in that they focus on accelerating the feature computation in point cloud DNNs. For instance, Point-X and Mesorasi exploit the spatial locality and computation redundancy, respectively. All three use brute-force neighbor search and, thus, can directly benefit from the optimizations (approximate neighbor search and selective bank conflict elision) proposed in this paper. We show 1.9 $\times$ speedup and 36% energy reduction over Mesorasi in Sec. 9.5.2.

Neighbor Search This paper targets neighbor search in low-dimensional space (2/3D), which is a fundamental building block in many computational science and engineering fields, where physical objects naturally lie in 2/3D space, such as computational fluid dynamics (Ihmsen et al., 2011), computer graphics (Yifan et al., 2019), and vision (Xu et al., 2019; Lu et al., 2021). Prior work has explored both algorithmic and hardware solutions to accelerate neighbor search (Heinzle et al., 2008; Qiu et al.,

2009; Aly et al., 2011; Kuhara et al., 2013; Winterstein et al., 2013; Gieseke et al., 2014; Xu et al., 2019; Pinkham et al., 2020b), many of which are approximate in their nature (Miclet and Dabouz, 1983; Arya et al., 1998; Ma and McCool, 2002; Greenspan and Yurick, 2003; Purcell et al., 2005; Heinzle et al., 2008). We provide a quantitative comparison with QuickNN (Pinkham et al., 2020b) and Tigris (Xu et al., 2019), two most relevant accelerators in Sec. 9.5.5.

Optimizing Irregular Memory Accesses Recent work has made significant strides in domain-agnostic prefetching for irregular applications (Ainsworth and Jones, 2018; Talati et al., 2021; Naithani et al., 2021). Our split-tree structure can be seen as an application-specific prefetcher and achieves "perfect prefetching" in that 1) off-chip data accesses are overlapped with computation, 2) data needed by the accelerator are readily available on-chip without stalls, and 3) no redundant DRAM accesses are needed.

Our split-tree structure also serves as an irregular tiling strategy, akin to propagation blocking for graph algorithms (Beamer et al., 2017), but the decision as to which partition (sub-tree) a point is stored is based on the geometric position of a point.

Approximation Techniques Our approximation techniques exploit the inexact nature of DNNs. Selective bank conflict elision can be seen as a form of value approximation, bearing similarity to such approximation in general-purpose processors (San Miguel et al., 2014; Miguel et al., 2015; Rengasamy et al., 2015; San Miguel et al., 2016; Wong et al., 2016). However, different from prior systems where the accuracy control is empirical, we integrate approximation into the training process; this allows us to provide statistical accuracy guarantees.

10 Retrospective and Prospective Remarks

10.1 Retrospective

Mobile vision system plays an even more important role in many applications, including autonomous devices, AR/VR, and IoT infrastructures. Meanwhile, these applications are also facing numerous challenges in terms of real-time performance and energy efficiency due to the limited computing resources on these devices. My thesis provides the first step to address these challenges, by exploiting compute redundancies in vision systems and hardware-algorithm co-designs. These core ideas in my thesis open up new possibilities for a wide range of mobile vision systems. The following paragraphs summarize the new ideas that carry through my thesis.

Exploiting Computation Redundancies Real-time performance is crucial for many mobile applications. However, today's mobile devices often lack the computation resources to support these applications. My thesis demonstrates that there are significant redundancies during the application execution that can be exploited to improve the overall application performance. Specifically, EDGAZE and ASV both explore the input redundancies across temporally-adjacent frames to effectively amortize the computation cost. EDGAZE leverages the temporal input redundancy to narrow down the ROI for subsequent processing for adjacent frames, while ASV exploits the sim-

ilarities across multiple input frames to extrapolate the vision algorithm's result with lightweight computation. Additionally, MESORASI leverages the "seemingly" redundant computations in point cloud algorithms themselves and approximates the original algorithms to maximize the parallelism.

Aware Hardware Constraints and Capabilities For any use cases in mobile applications, it is important for the vision algorithm designer to be aware of the implications of device constraints on their algorithm performance and to design algorithms that can accommodate such hardware limitations accordingly. These constraints include memory bandwidth, computation budget, and other factors. On the other hand, algorithm design should also exploit the hardware capabilities to achieve better performance and energy efficiency. EDGAZE takes the advantages of in-sensor computing capabilities of today's CIS and proposes an algorithm that utilizes the compute resources inside CIS while being aware of the compute budgets and overheads. Meanwhile, ASV considers hardware constraints such as memory bandwidth, on-chip buffer size, and process units, and formulates its dataflow optimization as a constrained optimization to better utilize the resources on a mobile SoC.

Architectural Generality vs. Domain Specialization As we move into the post-Moore's law era, hardware performance is no longer doubling every two years. However, the demand for improving performance in many emerging applications such as AR/VR and IoT devices, continues to grow. To meet such a goal, hardware communities are turning to domain specialization. However, inventing new hardware is a lengthy process with an even longer adoption period. An alternative to inventing brand-new architectures is to propose hardware augmentations which require minimum overheads and build upon existing hardware designs. Both MESORASI and CRESCENT take this approach. MESORASI proposes minimum hardware supports on existing mobile SoC with a gathering unit to support reduction operation in deep point cloud analytics. Similarly, CRESCENT augments existing SRAM back design to eliminate bank conflicts. The key to enabling emerging applications with such small hardware augmentations

is hardware-algorithm co-design. By understanding the key bottlenecks of the applications, hardware and algorithm optimization are co-designed to achieve even better application performance than can be achieved with brand-new hardware alone.

10.2 Prospective

The field of mobile visual computing is constantly evolving and has tremendous potential to unlock many impossibilities. As the demand for mobile vision systems continues to grow, it is important to outline several key challenges and opportunities that lie ahead.

Towards Domain-Specific Sensor Architecture Undoubtedly, in-sensor computing is a promising direction toward real-time, energy-efficient mobile visual systems. One of the major advantages of in-sensor computing is the reduction of the overall data bandwidth and transmission overhead, leading to an improvement in the overall system efficiency. As the number of sensors on mobile vision systems increases, it is more likely that the sensors' functionality becomes highly specialized. Therefore, designing domain-specific sensor architecture can perform real-time analysis of visual data in-situ without the need for external processing, allowing for faster and more responsive systems.

Privacy-Preserved Mobile Vision Privacy preservation is an important consideration in mobile vision systems, particularly as many applications become more ubiquitous in our daily lives. With more emerging applications relying on various sensors for visual perception, it is important to protect user privacy and prevent the leakage of sensitive information. As such, privacy-preserved mobile vision becomes an active research direction to improve the reliability and robustness of today's vision systems.

Compulsory Approximation of Deep Learning Analytics Today's mobile vision systems rely heavily on deep learning algorithms to perform various vision tasks and make accurate predictions. However, these algorithms are often deep learning algorithms which require intensive computation and exhibit irregular memory access. Thus,

it is challenging to perform efficient training and inference using today's hardware. One promising solution is to approximate the original deep learning algorithms since exact computation is often not necessary due to the approximate nature of deep learning methods. By exploiting such characteristics of deep learning methods and modifying deep learning computation to be hardware-friendly, it becomes easier to execute them on today's existing hardware. Meanwhile, exposing necessary hardware details during approximate deep-learning training can further improve the algorithm's accuracy and energy efficiency. By pursuing this direction, we can unlock the full potential of deep learning while achieving better system performance and efficiency.

Bibliography

Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K Jha. 2009. GARNET: A detailed on-chip network model inside a full-system simulator. In *2009 IEEE international symposium on performance analysis of systems and software*. IEEE, 33–42.

Sam Ainsworth and Timothy M Jones. 2018. An event-triggered programmable prefetcher for irregular workloads. *ACM Sigplan Notices* 53, 2 (2018), 578–592.

Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh K Gupta, and Hadi Esmaeilzadeh. 2018. Snapea: Predictive early activation for reducing computation in deep convolutional neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 662–673.

Cuneyt Akinlar, Hatice Kubra Kucukkartal, and Cihan Topal. 2021. Accurate CNN-based pupil segmentation with an ellipse fit error regularization term. *Expert Systems with Applications* (2021), 116004.

J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture*.

Pierre Alliez, François Forge, Livio De Luca, Marc Pierrot-Deseilligny, and Marius Preda. 2017. Culture 3D Cloud: A Cloud Computing Platform for 3D Scanning, Documentation, Preservation and Dissemination of Cultural Heritage. (2017).

Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN Accelerators. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*.

Mohamed Aly, Mario Munich, and Pietro Perona. 2011. Distributed kd-trees for retrieval from very large image collections. In *Proceedings of the British machine vision conference (BMVC)*, Vol. 17.

Anastasios N Angelopoulos, Julien NP Martel, Amit PS Kohli, Jorg Conradt, and Gordon Wetzstein. 2020. Event based, near eye gaze tracking beyond 10,000 Hz. *arXiv preprint arXiv:2004.03577* (2020).

Arm. 2016. Artisan Memory Compilers. https://developer.arm.com/ip-products/physical-ip/embedded-memory.

Arm. 2018. ARM's First Generation ML Processor, HotChips 30. https://www.hotchips.org/hc30/2conf/2.07_ARM_ML_Processor_HC30_ARM_2018_08_17.pdf

Sunil Arya, David M Mount, Nathan S Netanyahu, Ruth Silverman, and Angela Y Wu. 1998. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)* 45, 6 (1998), 891–923.

Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25.

Scott Beamer, Krste Asanović, and David Patterson. 2017. Reducing pagerank communication via propagation blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 820–831.

Jens Behley, Martin Garbade, Andres Milioto, Jan Quenzel, Sven Behnke, Cyrill Stachniss, and Jurgen Gall. 2019. SemanticKITTI: A dataset for semantic scene understanding of lidar sequences. In *Proceedings of the 13th IEEE International Conference on Computer Vision*.

Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

Thierry M Bernard, Bertrand Y Zavidovique, and Francis J Devos. 1993. A programmable artificial retina. *IEEE Journal of Solid-State Circuits* 28, 7 (1993), 789–798.

M Bigas, Enric Cabruja, Josep Forest, and Joaquim Salvi. 2006. Review of CMOS image sensors. *Microelectronics journal* 37, 5 (2006), 433–451.

Kyeongryeol Bong, Sungpill Choi, Changhyeon Kim, Donghyeon Han, and Hoi-Jun Yoo. 2017a. A low-power convolutional neural network face recognition processor and a CIS integrated with always-on face detector. *IEEE Journal of Solid-State Circuits* 53, 1 (2017), 115–123.

Kyeongryeol Bong, Sungpill Choi, Changhyeon Kim, Sanghoon Kang, Youchang Kim, and Hoi-Jun Yoo. 2017b. 14.6 A 0.62 mW ultra-low-power convolutional-neural-network face-recognition processor and a CIS integrated with always-on haar-like face detector. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 248–249.

David Brooks, Vivek Tiwari, and Margaret Martonosi. 2000. Wattch: A framework for architectural-level power analysis and optimizations. *ACM SIGARCH Computer Architecture News* 28, 2 (2000), 83–94.

M. Z. Brown, D. Burschka, and G. D. Hager. 2003. Advances in computational stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Mark Buckler, Philip Bedoukian, Suren Jayasuriya, and Adrian Sampson. 2018. EVA^2: Exploiting Temporal Redundancy in Live Computer Vision. In *Proceedings of the 45th ACM/IEEE Annual International Symposium on Computer Architecture*.

Marco Caligari, Marco Godi, Simone Guglielmetti, Franco Franchignoni, and Antonio Nardone. 2013. Eye tracking communication devices in amyotrophic lateral sclerosis: impact on disability and quality of life. *Amyotrophic Lateral Sclerosis and Frontotemporal Degeneration* 14, 7-8 (2013), 546–552.

John Canny. 1986. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence* 6 (1986), 679–698.

Weidong Cao, Yilong Zhao, Adith Boloor, Yinhe Han, Xuan Zhang, and Li Jiang. 2021. Neural-PIM: Efficient processing-in-memory with neural approximation of peripherals. *IEEE Trans. Comput.* 71, 9 (2021), 2142–2155.

Raffaele Capoccia, Assim Boukhayma, and Christian Enz. 2020. Experimental Verification of the Impact of Analog CMS on CIS Readout Noise. *IEEE Transactions on Circuits and Systems I: Regular Papers* 67, 3 (2020), 774–784.

Jan Cech, Jordi Sanchez-Riera, and Radu P. Horaud. 2011. Scene Flow Estimation by Growing Correspondence Seeds. In *Proceedings of the 21st IEEE Conference on Computer Vision and Pattern Recognition*.

Yang Chai. 2020. In-sensor computing for machine vision.

Karthik Chandrasekar, Christian Weis, Yonghui Li, Benny Akesson, Norbert Wehn, and Kees Goossens. 2012. DRAMPower: Open-source DRAM power & energy estimation tool. *URL: http://www. drampower. info* 22 (2012).

Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. 2015. *ShapeNet: An Information-Rich 3D Model Repository*.

Technical Report arXiv:1512.03012 [cs.GR]. Stanford University — Princeton University — Toyota Technological Institute at Chicago.

Jia-Ren Chang and Yong-Sheng Chen. 2018. Pyramid Stereo Matching Network. In *Proceedings of 28th IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

Aayush K Chaudhary, Rakshit Kothari, Manoj Acharya, Shusil Dangi, Nitinraj Nair, Reynold Bailey, Christopher Kanan, Gabriel Diaz, and Jeff B Pelz. 2019. RITnet: Real-time semantic segmentation of the eye for gaze tracking. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. IEEE, 3698–3702.

Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014a. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Y. Chen, J. Emer, and V. Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture*.

Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. 2014b. DaDianNao: A Machine-Learning Supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*.

Jaihyuk Choi, Sungjae Lee, Youngdoo Son, and Soo Youn Kim. 2020. Design of an always-on image sensor using an analog lightweight convolutional neural network. *Sensors* 20, 11 (2020), 3101.

Christopher Choy, JunYoung Gwak, and Silvio Savarese. 2019. 4d spatio-temporal convnets: Minkowski convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 3075–3084.

Navneet Dalal and Bill Triggs. 2005. Histograms of oriented gradients for human detection. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, Vol. 1. Ieee, 886–893.

Gisler Damian, Tobi Delbruck, and Patrick Lichtsteiner. 2007. Eye Tracking using Event-Based Silicon Retina. (2007).

I. Daribo, R. Furukawa, R. Sagawa, H. Kawasaki, S. Hiura, and N. Asada. 2011. Point cloud compression for grid-pattern-based 3D scanning system. In *2011 Visual Communications and Image Processing (VCIP)*.

Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (1974), 256–268.

Kai Dierkes, Moritz Kassner, and Andreas Bulling. 2019. A fast approach to refraction-aware eye-model fitting and gaze prediction. In *Proceedings of the 11th ACM Symposium on Eye Tracking Research & Applications*. 1–9.

Xin Dong, Barbara De Salvo, Meng Li, Chiao Liu, Zhongnan Qu, HT Kung, and Ziyun Li. 2022. SplitNets: Designing Neural Architectures for Efficient Distributed Computing on Head-Mounted Systems. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 12559–12569.

Lisa Eadicicco. 2022. Apple says its new A13 Bionic chip brings hours of extra battery life to new iPhones. https://en.wikichip.org/wiki/apple/ax/a13

Ryoji Eki, Satoshi Yamada, Hiroyuki Ozawa, Hitoshi Kai, Kazuyuki Okuike, Hareesh Gowtham, Hidetomo Nakanishi, Edan Almog, Yoel Livne, Gadi Yuval, et al. 2021. 9.6 A 1/2.3 inch 12.3 Mpixel with on-chip 4.97 TOPS/W CNN processor back-illuminated stacked CMOS image sensor. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 64. IEEE, 154–156.

Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *kdd*, Vol. 96. 226–231.

Gunnar Farnebäck. 2002. *Polynomial expansion for orientation and motion estimation*. Ph.D. Dissertation. Linköping University Electronic Press.

Gunnar Farnebäck. 2003. Two-frame motion estimation based on polynomial expansion. In *Proceedings of the 13th Scandinavian Conference on Image Analysis*.

Yu Feng, Nathan Goulding-Hotta, Asif Khan, Hans Reyserhove, and Yuhao Zhu. 2022a. Real-Time Gaze Tracking with Event-Driven Eye Segmentation. In *2022 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. IEEE, 399–408.

Yu Feng, Gunnar Hammonds, Yiming Gan, and Yuhao Zhu. 2022b. Crescent: taming memory irregularities for accelerating deep point cloud analytics. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 962–977.

Yu Feng, Patrick Hansen, Paul N Whatmough, Guoyu Lu, and Yuhao Zhu. 2023. Fast and Accurate: Video Enhancement Using Sparse Depth. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 4492–4500.

Yu Feng, Shaoshan Liu, and Yuhao Zhu. 2020a. Real-time spatio-temporal lidar point cloud compression. In *2020 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE, 10766–10773.

Yu Feng, Boyuan Tian, Tiancheng Xu, Paul Whatmough, and Yuhao Zhu. 2020b. Mesorasi: Architecture support for point cloud analytics via delayed-aggregation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1037–1050.

Yu Feng, Paul Whatmough, and Yuhao Zhu. 2019. Asv: Accelerated stereo vision system. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 643–656.

Yu Feng and Yuhao Zhu. 2019. Pes: Proactive event scheduling for responsive and energy-efficient mobile web computing. In *Proceedings of the 46th International Symposium on Computer Architecture*. 66–78.

Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

Philipp Fischer, Alexey Dosovitskiy, Eddy Ilg, Philip Häusser, Caner Hazirbas, Vladimir Golkov, Patrick van der Smagt, Daniel Cremers, and Thomas Brox. 2015. FlowNet: Learning Optical Flow with Convolutional Networks. In *Proceedings of the 15th IEEE International Conference on Computer Vision*.

Martin A Fischler and Robert C Bolles. 1981. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM* (1981).

Wolfgang Fuhl, Thiago Santini, Gjergji Kasneci, and Enkelejda Kasneci. 2016a. Pupilnet: Convolutional neural networks for robust pupil detection. *arXiv preprint arXiv:1601.04902* (2016).

Wolfgang Fuhl, Thiago C Santini, Thomas Kübler, and Enkelejda Kasneci. 2016b. Else: Ellipse selection for robust pupil detection in real-world environments. In *Proceedings of the Ninth Biennial ACM Symposium on Eye Tracking Research & Applications*. 123–130.

Guillermo Gallego, Tobi Delbruck, Garrick Orchard, Chiara Bartolozzi, Brian Taba, Andrea Censi, Stefan Leutenegger, Andrew Davison, Jörg Conradt, Kostas Dani-

ilidis, et al. 2019. Event-based vision: A survey. *arXiv preprint arXiv:1904.08405* (2019).

Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. 2019. Tangram: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating System*.

Stephan J Garbin, Yiru Shen, Immo Schuetz, Robert Cavin, Gregory Hughes, and Sachin S Talathi. 2019. Openeds: Open eye dataset. *arXiv preprint arXiv:1905.03702* (2019).

Andreas Geiger. 2012. KITTI Stereo Benchmark. http://www.cvlibs.net/datasets/kitti/eval_scene_flow.php?benchmark=stereo

Andreas Geiger, Philip Lenz, and Raquel Urtasun. 2012a. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Proceedings of the 25th IEEE Conference on Computer Vision and Pattern Recognition*.

Andreas Geiger, Philip Lenz, and Raquel Urtasun. 2012b. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

Andreas Geiger, Martin Roser, and Raquel Urtasun. 2010. Efficient Large-Scale Stereo Matching. In *Proceedings of the 10th Asian Conference on Computer Vision*.

F. Gieseke, J. Heinermann, C. Oancea, and C. Igel. 2014. Buffer k-d trees: Processing massive nearest neighbor queries on GPUs. (2014).

Ross Girshick. 2015. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*. 1440–1448.

Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 580–587.

T. Golla and R. Klein. 2015. Real-time point cloud compression. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Jorge Gomez, Saavan Patel, Syed Shakib Sarwar, Ziyun Li, Raffaele Capoccia, Zhao Wang, Reid Pinkham, Andrew Berkovich, Tsung-Hsun Tsai, Barbara De Salvo, et al. 2022. Distributed On-Sensor Compute System for AR/VR Devices: A Semi-Analytical Simulation Framework for Power Estimation. *arXiv preprint arXiv:2203.07474* (2022).

Benjamin Graham, Martin Engelcke, and Laurens Van Der Maaten. 2018. 3d semantic segmentation with submanifold sparse convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 9224–9232.

Michael Greenspan and Mike Yurick. 2003. Approximate kd tree search for efficient ICP. In *Fourth International Conference on 3-D Digital Imaging and Modeling, 2003. 3DIM 2003. Proceedings.* IEEE, 442–448.

Markus Gross and Hanspeter Pfister. 2011. *Point-based graphics*. Elsevier.

Boris Grot, Joel Hestness, Stephen W Keckler, and Onur Mutlu. 2011. Kilo-NOC: a heterogeneous network-on-chip architecture for scalability and service guarantees. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 401–412.

Camare Working Group. 2021. MIPI White Paper: An Introductory Guide to MIPI Automotive SerDes Solutions (MASS). https://www.mipi.org/introductory-guide-to-mass

Eduardo Gudis, Pullan Lu, David Berends, Kevin Kaighn, Gooitzen Wal, Gregory Buchanan, Sek Chai, and Michael Piacentino. 2013. An embedded vision services framework for heterogeneous accelerators. In *Proceedings of the 23rd IEEE conference on computer vision and pattern recognition workshops*.

Yulan Guo, Hanyun Wang, Qingyong Hu, Hao Liu, Li Liu, and Mohammed Bennamoun. 2020. Deep learning for 3d point clouds: A survey. *IEEE transactions on pattern analysis and machine intelligence* (2020).

Matthew R Guthaus, James E Stine, Samira Ataei, Brian Chen, Bin Wu, and Mehedi Sarwar. 2016. OpenRAM: An open-source memory compiler. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–6.

William L. Hamilton. 2017. GraphSage in TensorFlow. https://github.com/williamleif/GraphSAGE

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016a. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 243–254.

S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. 2016b. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture*.

Dan Witzner Hansen and Arthur EC Pece. 2005. Eye tracking in the wild. *Computer Vision and Image Understanding* 98, 1 (2005), 155–181.

Richard Hartley and Andrew Zisserman. 2003. *Multiple view geometry in computer vision*. Cambridge university press.

Tsutomu Haruta, Tsutomu Nakajima, Jun Hashizume, Taku Umebayashi, Hiroshi Takahashi, Kazuo Taniguchi, Masami Kuroda, Hiroshi Sumihiro, Koji Enoki, Takatsugu Yamasaki, et al. 2017. 4.6 A 1/2.3 inch 20Mpixel 3-layer stacked CMOS Image Sensor with DRAM. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 76–77.

Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017a. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*. 2961–2969.

Yihui He, Xiangyu Zhang, and Jian Sun. 2017b. Channel Pruning for Accelerating Very Deep Neural Networks. In *Proceedings of the 16th IEEE International Conference on Computer Vision*.

James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.* 33, 4 (2014), 144–1.

Kartik Hegde, Rohit Agrawal, Yulun Yao, and Christopher W Fletcher. 2018. Morph: Flexible Acceleration for 3D CNN-Based Video Understanding. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*.

Simon Heinzle, Gaël Guennebaud, Mario Botsch, and Markus H. Gross. 2008. A Hardware Processing Unit for Point Sets. In *Acm Siggraph/eurographics Symposium on Graphics Hardware*.

Tomoki Hirata, Hironobu Murata, Hideaki Matsuda, Yojiro Tezuka, and Shiro Tsunai. 2021. 7.8 A 1-inch 17Mpixel 1000fps Block-Controlled Coded-Exposure Back-Illuminated Stacked CMOS Image Sensor for Computational Imaging and Adaptive Dynamic Range Control. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 64. IEEE, 120–122.

H. Hirschmuller. 2005. Accurate and efficient stereo processing by semi-global matching and mutual information. In *Proceedings of the 15th IEEE Conference on Computer Vision and Pattern Recognition*.

Sunpyo Hong and Hyesoon Kim. 2010. An integrated GPU power and performance model. In *Proceedings of the 37th annual international symposium on Computer architecture*. 280–289.

Berthold KP Horn and Brian G Schunck. 1981. Determining optical flow. *Artificial intelligence*.

Armin Hornung, Kai Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. 2013. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots* (2013).

JW Horton, RV Mazza, and H Dym. 1964. The scanistor—A solid-state image scanner. *Proc. IEEE* 52, 12 (1964), 1513–1528.

Tzu-Hsiang Hsu, Guan-Cheng Chen, Yi-Ren Chen, Chung-Chuan Lo, Ren-Shuo Liu, Meng-Fan Chang, Kea-Tiong Tang, and Chih-Cheng Hsieh. 2022. A 0.8 V Intelligent Vision Sensor with Tiny Convolutional Neural Network and Programmable Weights Using Mixed-Mode Processing-in-Sensor Technique for Image Classification. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. IEEE, 1–3.

Tzu-Hsiang Hsu, Yi-Ren Chen, Ren-Shuo Liu, Chung-Chuan Lo, Kea-Tiong Tang, Meng-Fan Chang, and Chih-Cheng Hsieh. 2020. A 0.5-V real-time computational CMOS image sensor with programmable kernel for feature extraction. *IEEE Journal of Solid-State Circuits* 56, 5 (2020), 1588–1596.

Chao-Tsung Huang, Yu-Chun Ding, Huan-Ching Wang, Chi-Wen Weng, Kai-Ping Lin, Li-Wei Wang, and Li-De Chen. 2019. eCNN: A Block-Based and Highly-Parallel

CNN Accelerator for Edge Inference. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

Yan Huang, Jingliang Peng, C.-C. Jay Kuo, and M. Gopi. 2006. Octree-Based Progressive Geometry Coding of Point Clouds. In *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*.

Muhammad Huzaifa, Rishi Desai, Samuel Grayson, Xutao Jiang, Ying Jing, Jae Lee, Fang Lu, Yihan Pang, Joseph Ravichandran, Finn Sinclair, et al. 2021. ILLIXR: Enabling End-to-End Extended Reality Research. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 24–38.

Bongjoon Hyun, Jiwon Lee, and Minsoo Rhu. 2021. Characterization and Analysis of Deep Learning for 3D Point Cloud Analytics. *IEEE Computer Architecture Letters* 20, 2 (2021), 106–109.

Markus Ihmsen, Nadir Akinci, Markus Becker, and Matthias Teschner. 2011. A parallel SPH implementation on multi-core CPUs. In *Computer Graphics Forum*, Vol. 30. Wiley Online Library, 99–112.

Intel. 2017. Intel Movidius Myriad X VPU. https://www.movidius.com/myriadx

IRDS. 2022. International Roadmap for Devices and Systems. https://irds.ieee.org/.

ITU. 2020. Information technology — JPEG 2000 image coding system. https://www.itu.int/rec/T-REC-T.800/

M Jakubowski and G Pastuszak. 2013. Block-based Motion Estimation Algorithms–A Survey. *Opto-Electronics Review*.

E. S. Jang, M. Preda, K. Mammou, A. M. Tourapis, J. Kim, D. B. Graziosi, S. Rhyu, and M. Budagavi. 2019. Video-Based Point-Cloud-Compression Standard in MPEG: From Evidence Collection to Committee Draft [Standards in a Nutshell]. *IEEE Signal Processing Magazine* (2019).

Paul G Jespers. 2010. *The gm/ID Methodology, a sizing tool for low-voltage analog CMOS Circuits*. Springer.

Bengt E Jonsson. 2010. A survey of A/D-Converter performance evolution. In *2010 17th IEEE International Conference on Electronics, Circuits and Systems*. IEEE, 766–769.

M Jose and Rocio Del Rio. 2013. CMOS sigma-delta converters: practical design guide. (2013).

Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017b. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual ACM/IEEE International Symposium on Computer Architecture*.

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi

Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017a. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proc. of ISCA*.

J. Kammerl, N. Blodow, R. B. Rusu, S. Gedikli, M. Beetz, and E. Steinbach. 2012. Real-time compression of point cloud streams. In *2012 IEEE International Conference on Robotics and Automation*.

Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G Rogers, Tor M Aamodt, and Nikos Hardavellas. 2021. AccelWattch: A Power Modeling Framework for Modern GPUs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 738–753.

Anuradha Kar and Peter Corcoran. 2017. A review and analysis of eye-gaze estimation systems, algorithms and performance evaluation methods in consumer platforms. *IEEE Access* 5 (2017), 16495–16519.

Amandeep Kaur, Deepak Mishra, KM Amogh, and Mukul Sarkar. 2020. On-array compressive acquisition in cmos image sensors using accumulated spatial gradients. *IEEE Transactions on Circuits and Systems for Video Technology* 31, 2 (2020), 523–532.

Alex Kendall, Hayk Martirosyan, Saumitro Dasgupta, Peter Henry, Ryan Kennedy, Abraham Bachrach, and Adam Bry. 2017. End-to-End Learning of Geometry and Context for Deep Stereo Regression. In *Proceedings of 27th IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

Hanme Kim, Stefan Leutenegger, and Andrew J Davison. 2016. Real-time 3D re-

construction and 6-DoF tracking with an event camera. In *European Conference on Computer Vision*. Springer, 349–364.

Soo-Hyung Kim, Guee-Sang Lee, Hyung-Jeong Yang, et al. 2019. Eye semantic segmentation with a lightweight model. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. IEEE, 3694–3697.

Seong-Jin Kim, Kwang-Hyun Lee, Sang-Wook Han, and Euisik Yoon. 2005. A 200/spl times/160 pixel CMOS fingerprint recognition SoC with adaptable column-parallel processors. In *ISSCC. 2005 IEEE International Digest of Technical Papers. Solid-State Circuits Conference, 2005.* IEEE, 250–596.

David B Kirk and W Hwu Wen-Mei. 2016. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann.

Christof Koch and Hua Li. 1994. *Vision chips: implementing vision algorithms with analog VLSI circuits*. IEEE computer society press.

Venkatesh Kodukula, Saad Katrawala, Britton Jones, Carole-Jean Wu, and Robert LiKamWa. 2021a. Dynamic temperature management of near-sensor processing for energy-efficient high-fidelity imaging. *Sensors* 21, 3 (2021), 926.

Venkatesh Kodukula, Alexander Shearer, Van Nguyen, Srinivas Lingutla, Yifei Liu, and Robert LiKamWa. 2021b. Rhythmic pixel regions: multi-resolution visual sensing system towards high-precision visual computing at low power. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 573–586.

Tao Kong, Anbang Yao, Yurong Chen, and Fuchun Sun. 2016. Hypernet: Towards accurate region proposal generation and joint object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 845–853.

Rakshit S Kothari, Aayush K Chaudhary, Reynold J Bailey, Jeff B Pelz, and Gabriel J Diaz. 2021. EllSeg: An Ellipse Segmentation Framework for Robust Gaze Tracking. *IEEE Transactions on Visualization and Computer Graphics* 27, 5 (2021), 2757–2767.

M. Krivokuća, P. A. Chou, and M. Koroteev. 2020. A Volumetric Approach to Point Cloud Compression–Part II: Geometry Compression. *IEEE Transactions on Image Processing* (2020).

Takuya Kuhara, Takaaki Miyajima, Masato Yoshimi, and Hideharu Amano. 2013. *An FPGA Acceleration for the Kd-tree Search in Photon Mapping*.

Oichi Kumagai, Atsumi Niwa, Katsuhiko Hanzawa, Hidetaka Kato, Shinichiro Futami, Toshio Ohyama, Tsutomu Imoto, Masahiko Nakamizo, Hirotaka Murakami, Tatsuki Nishino, et al. 2018. A 1/4-inch 3.9 Mpixel low-power event-driven back-illuminated stacked CMOS image sensor. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 86–88.

H.T. Kung, Bradley McDanel, and Sai Qian Zhang. 2019. Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining Under Joint Optimization. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Hyoukjun Kwon, Krishnakumar Nair, Jamin Seo, Jason Yik, Debabrata Mohapatra, Dongyuan Zhan, Jinook Song, Peter Capak, Peizhao Zhang, Peter Vajda, et al. 2022. XRBench: An Extended Reality (XR) Machine Learning Benchmark Suite for the Metaverse. *arXiv preprint arXiv:2211.08675* (2022).

Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

Minho Kwon, Seunghyun Lim, Hyeokjong Lee, Il-Seon Ha, Moo-Young Kim, Il-Jin Seo, Suho Lee, Yongsuk Choi, Kyunghoon Kim, Hansoo Lee, et al. 2020. A Low-Power 65/14nm Stacked CMOS Image Sensor. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–4.

Sébastien Lasserre, David Flynn, and Shouxing Qu. 2019. Using Neighbouring Nodes for the Compression of Octrees Representing the Geometry of Point Clouds. In *Proceedings of the 10th ACM Multimedia Systems Conference*.

E. Lauwers and G. Gielen. 2002. Power estimation methods for analog circuits for architectural exploration of integrated systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 10, 2 (2002), 155–162.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

Edward H. Lee and S. Simon Wong. 2017. Analysis and Design of a Passive Switched-Capacitor Matrix Multiplier for Approximate Computing. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 261–271.

Frank Lee. 2017. Choose the Right 3D Vision Camera For Your IoT Device. https://medium.com/iotforall/choose-the-right-3d-vision-camera-for-your-iot-device-962d95c581cb

S. K. Lee, P. N. Whatmough, D. Brooks, and G. Wei. 2019. A 16-nm Always-On DNN Processor With Adaptive Clocking and Multi-Cycle Banked SRAMs. *IEEE Journal of Solid-State Circuits*.

Ville V Lehtola, Harri Kaartinen, Andreas Nüchter, Risto Kaijaluoto, Antero Kukko, Paula Litkey, Eija Honkavaara, Tomi Rosnell, Matti T Vaaja, Juho-Pekka Virtanen, et al. 2017. Comparison of the selected state-of-the-art 3D indoor scanning and point cloud generation methods. *Remote sensing* 9, 8 (2017), 796.

Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling energy optimizations in GPGPUs. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 487–498.

Yue Leng, Chi-Chun Chen, Qiuyue Sun, Jian Huang, and Yuhao Zhu. 2019. Energy-efficient video processing for virtual reality. In *Proceedings of the 46th International Symposium on Computer Architecture*.

Marc Levoy and Turner Whitted. 1985. *The use of points as a display primitive*. Citeseer.

Haitong Li, Mudit Bhargava, Paul N. Whatmough, and H-S Philip Wong. 2019. On-Chip Memory Technology Design Space Explorations for Mobile Deep Neural Network Accelerators. In *Proceedings of the 56th Annual Design Automation Conference*.

Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*. 469–480.

Robert LiKamWa, Yunhui Hou, Julian Gao, Mia Polansky, and Lin Zhong. 2016. Redeye: analog convnet image sensor architecture for continuous mobile vision. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 255–266.

Robert LiKamWa, Bodhi Priyantha, Matthai Philipose, Lin Zhong, and Paramvir Bahl. 2013. Energy characterization and optimization of image sensing toward continuous mobile vision. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. 69–82.

Byong Chan Lim and Mark Horowitz. 2019. An Analog Model Template Library: Simplifying Chip-Level, Mixed-Signal Design Verification. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 1 (2019), 193–204.

Yujun Lin, Zhekai Zhang, Haotian Tang, Hanrui Wang, and Song Han. 2021. PointAcc: Efficient Point Cloud Accelerator. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 449–461.

Chiao Liu, Andrew Berkovich, Song Chen, Hans Reyserhove, Syed Shakib Sarwar, and Tsung-Hsun Tsai. 2019a. Intelligent Vision Systems–Bringing Human-Machine Interface to AR/VR. In *2019 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 10–5.

Chiao Liu, Song Chen, Tsung-Hsun Tsai, Barbara De Salvo, and Jorge Gomez. 2022. Augmented Reality-The Next Frontier of Image Sensors and Compute Systems. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. IEEE, 426–428.

Hongche Liu, Tsai-Hong Hong, Martin Herman, Ted Camus, and Rama Chellappa. 1998. Accuracy vs efficiency trade-offs in optical flow algorithms. *Computer vision and image understanding*.

Yongcheng Liu, Bin Fan, Gaofeng Meng, Jiwen Lu, Shiming Xiang, and Chunhong Pan. 2019b. DensePoint: Learning densely contextual representation for efficient point cloud processing. In *Proceedings of the 14th IEEE International Conference on Computer Vision*.

Zhijian Liu, Haotian Tang, Yujun Lin, and Song Han. 2019c. Point-Voxel CNN for efficient 3D deep learning. In *Advances in Neural Information Processing Systems*. 963–973.

Yawen Lu, Yuhao Zhu, and Guoyu Lu. 2021. 3D SceneFlowNet: Self-Supervised 3D Scene Flow Estimation Based on Graph CNN. In *2021 IEEE International Conference on Image Processing (ICIP)*. IEEE, 3647–3651.

Bruce D Lucas and Takeo Kanade. 1981. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*.

Tianrui Ma, Weidong Cao, Fei Qiao, Ayan Chakrabarti, and Xuan Zhang. 2022. HOG-Eye: neural approximation of hog feature extraction in rram-based 3d-stacked image sensors. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. 1–6.

Tianrui Ma, Yu Feng, Xuan Zhang, and Yuhao Zhu. 2023. CamJ: Enabling System-Level Energy Modeling and Architectural Exploration for In-Sensor Visual Computing. *arXiv preprint arXiv:2304.03320* (2023).

Vincent CH Ma and Michael D McCool. 2002. Low latency photon mapping using block hashing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Eurographics Association, 89–99.

Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2017. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In *Proceedings of the 25th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.

Mostafa Mahmoud, Kevin Siu, and Andreas Moshovos. 2018. Diffy: a Déjà vu-Free Differential Deep Neural Network Accelerator. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*.

Mostafa Mahmoud, Bojian Zheng, Alberto Delmás Lascorz, Felix HeideJonathan Assouline, Jonathan Assouline, Paul Boucher, Emmanuel Onzon, and Andreas

Moshovos. 2017. IDEAL: Image denoising accelerator. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*.

Nikolaus Mayer, Eddy Ilg, Philip Häusser, Philipp Fischer, Daniel Cremers, Alexey Dosovitskiy, and Thomas Brox. 2016. A Large Dataset to Train Convolutional Networks for Disparity, Optical Flow, and Scene Flow Estimation. In *Proceedings of 26th IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

Amrita Mazumdar, Thierry Moreau, Sung Kim, Meghan Cowan, Armin Alaghi, Luis Ceze, Mark Oskin, and Visvesh Sathe. 2017. Exploring computation-communication tradeoffs in camera systems. In *Proceedings of the 13th IEEE International Symposium on Workload Characterization*.

M. Menze and A. Geiger. 2015. Object scene flow for autonomous vehicles. In *Proceedings of the 25th IEEE Conference on Computer Vision and Pattern Recognition*.

Meta. 2019. OpenEDS 2019 Challenge. https://research.fb.com/programs/openeds-challenge/.

Donald Michie. 1968. "Memo" functions and machine learning. *Nature*.

Laurent Miclet and Mhamed Dabouz. 1983. Approximative fast nearest-neighbour recognition. *Pattern Recognition Letters* 1, 5-6 (1983), 277–285.

Micron. 2014. Micron 178-Ball, Single-Channel Mobile LPDDR3 SDRAM Features. https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/mobile-dram/low-power-dram/lpddr3/178b_8-16gb_2c0f_mobile_lpddr3.pdf

Micron. 2020. Micron System Power Calculators. https://www.micron.com/support/tools-and-utilities/power-calc

Microsoft. 2017. Second Version of HoloLens HPU will Incorporate AI Co-processor for Implementing DNNs. https://www.microsoft.com/

en-us/research/blog/second-version-hololens-hpu-will-incorporate-ai-coprocessor-implementing-dnns/

Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. 2015. Doppelgänger: a cache for approximate computing. In *Proceedings of the 48th International Symposium on Microarchitecture*. 50–61.

Vicente Morell, Sergio Orts, Miguel Cazorla, and Jose Garcia-Rodriguez. 2014. Geometric 3D point cloud compression. *Pattern Recognition Letters* (2014).

SR Morrison. 1963. A new type of photosensitive junction device. *Solid-State Electronics* 6, 5 (1963), 485–494.

Burhan Ahmad Mudassar, Priyabrata Saha, Yun Long, Muhammad Faisal Amir, Evan Gebhardt, Taesik Na, Jong Hwan Ko, Marilyn Wolf, and Saibal Mukhopadhyay. 2019. A camera with brain–embedding machine learning in 3d sensors. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 680–685.

Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. In *Proceedings of the 35th International Conference on Computer Graphics and Interactive Techniques*.

Hirotaka Murakami, Eric Bohannon, John Childs, Grace Gui, Eric Moule, Katsuhiko Hanzawa, Tomofumi Koda, Chiaki Takano, Toshimasa Shimizu, Yuki Takizawa, et al. 2022. A 4.9 Mpixel Programmable-Resolution Multi-Purpose CMOS Image Sensor for Computer Vision. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. IEEE, 104–106.

B. Murmann. 2022. ADC Performance Survey 1997-2022. http://web.stanford.edu/~murmann/adcsurvey.html.

Nachiappan Chidambaram Nachiappan, Haibo Zhang, Jihyun Ryoo, Niranjan Soundararajan, Anand Sivasubramaniam, Mahmut T Kandemir, Ravi Iyer, and Chita R Das. 2016. VIP: virtualizing IP chains on handheld platforms. *ACM SIGARCH Computer Architecture News* 43, 3 (2016), 655–667.

Ajeya Naithani, Sam Ainsworth, Timothy M Jones, and Lieven Eeckhout. 2021. Vector runahead. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 195–208.

Pauline C Ng and Steven Henikoff. 2003. SIFT: Predicting amino acid changes that affect protein function. *Nucleic acids research* 31, 13 (2003), 3812–3814.

Yves Nievergelt. 1994. Total least squares: State-of-the-art regression in numerical analysis. *SIAM review* (1994).

Nvidia. 2017a. Nvidia GP10B Spec. https://www.techpowerup.com/gpu-specs/nvidia-gp10b.g856

Nvidia. 2017b. NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge. https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/

Nvidia. 2017c. NVIDIA Jetson TX2 Module. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/

Nvidia. 2018. NVIDIA Reveals Xavier SOC Details. https://www.forbes.com/sites/moorinsights/2018/08/24/nvidia-reveals-xavier-soc-details/amp/

Nvidia. 2019. Nvidia Jetson AGX Xavier. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/.

Tilo Ochotta and Dietmar Saupe. 2004. Compression of Point-Based 3D Models by Shape-Adaptive Wavelet Coding of Multi-Height Fields. In *Proceedings of the First Eurographics Conference on Point-Based Graphics*.

OmniVision. 2021. OmniVision OV9782. https://www.ovt.com/sensors/OV9782.

OnSemi. 2015. OnSemi NOIP1SN025KA Datasheet. https://www.onsemi.com/pdf/datasheet/noip1sn025ka-d.pdf.

Cristina Palmero, Abhishek Sharma, Karsten Behrendt, Kapil Krishnakumar, Oleg V Komogortsev, and Sachin S Talathi. 2020. OpenEDS2020: open eyes dataset. *arXiv preprint arXiv:2005.03876* (2020).

Chanmin Park, Wenda Zhao, Injun Park, Nan Sun, and Youngcheol Chae. 2021. A 51-pJ/pixel 33.7-dB PSNR 4× compressive CMOS image sensor with column-parallel single-shot compressive sensing. *IEEE Journal of Solid-State Circuits* 56, 8 (2021), 2503–2515.

Asavari Patil. 2020. 3D Camera Market is Expected to Reach $7.6 Billion, Globally, by 2020. https://www.alliedmarketresearch.com/press-release/global-3D-camera-market-is-expected-to-reach-7-6-billion-by-2020-allied-market-research.html

Lillian Pentecost, Alexander Hankin, Marco Donato, Mark Hempstead, Gu-Yeon Wei, and David Brooks. 2022. NVMExplorer: A Framework for Cross-Stack Comparisons of Embedded Non-Volatile Memories. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 938–956.

Hanspeter Pfister, Matthias Zwicker, Jeroen Van Baar, and Markus Gross. 2000. Surfels: Surface elements as rendering primitives. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 335–342.

Reid Pinkham, Tanner Schmidt, and Andrew Berkovich. 2020a. Algorithm-aware neural network based image compression for high-speed imaging. In *2020 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR)*. IEEE, 196–199.

Reid Pinkham, Shuqing Zeng, and Zhengya Zhang. 2020b. Quicknn: Memory and performance optimization of kd tree based nearest neighbor search for 3d point clouds. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 180–192.

Matt Poremba, Sparsh Mittal, Dong Li, Jeffrey S Vetter, and Yuan Xie. 2015. Destiny: A tool for modeling emerging 3d nvm and edram caches. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1543–1546.

William Pugh and Tim Teitelbaum. 1989. Incremental computation via function caching. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*.

Timothy J Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. 2005. Photon mapping on programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses*. ACM, 258.

Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A Horowitz. 2013. Convolution engine: balancing efficiency & flexibility in specialized computing. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 24–35.

Charles R Qi, Wei Liu, Chenxia Wu, Hao Su, and Leonidas J Guibas. 2018. Frustum pointnets for 3d object detection from rgb-d data. In *Proceedings of the 31st IEEE Conference on Computer Vision and Pattern Recognition*. 918–927.

Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. 2017a. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 652–660.

Charles R Qi, Hao Su, Matthias Nießner, Angela Dai, Mengyuan Yan, and Leonidas J Guibas. 2016. Volumetric and multi-view cnns for object classification on 3d data. In *Proceedings of the 29th IEEE conference on computer vision and pattern recognition*.

Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. 2017b. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In *Advances in neural information processing systems*. 5099–5108.

Deyuan Qiu, Stefan May, and Andreas Nüchter. 2009. GPU-Accelerated Nearest Neighbor Search for 3D Registration. In *Proceedings of the 9th International Conference on Computer Vision Systems*.

Rajeev Ramanath, Wesley E Snyder, Youngjun Yoo, and Mark S Drew. 2005. Color image processing pipeline. *IEEE Signal processing magazine* 22, 1 (2005), 34–43.

Bharath Ramesh, Shihao Zhang, Zhi Wei Lee, Zhi Gao, Garrick Orchard, and Cheng Xiang. 2018. Long-term object tracking with a moving event camera.. In *Bmvc*. 241.

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems* 28 (2015), 91–99.

Prasanna Venkatesh Rengasamy, Anand Sivasubramaniam, Mahmut T Kandemir, and Chita R Das. 2015. Exploiting staleness for approximating loads on CMPs. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 343–354.

Market Research Report. 2022. Augmented Reality and Virtual Reality Market. https://www.marketsandmarkets.com/Market-Reports/augmented-reality-virtual-reality-market-1185.html

Gernot Riegler, Ali Osman Ulusoy, and Andreas Geiger. 2017. Octnet: Learning deep 3d representations at high resolutions. In *Proceedings of the 30th IEEE Conference on Computer Vision and Pattern Recognition*.

Marc Riera, Jose-Maria Arnau, and Antonio González. 2018. Computation reuse in DNNs by exploiting input similarity. In *Proceedings of the 45th IEEE Annual International Symposium on Computer Architecture*.

Szymon Rusinkiewicz and Marc Levoy. 2000. QSplat: A multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 343–352.

Radu Bogdan Rusu, Nico Blodow, and Michael Beetz. 2009. Fast point feature histograms (FPFH) for 3D registration. In *Proceedings of the 22nd IEEE International Conference on Robotics and Automation*.

Radu Bogdan Rusu and Steve Cousins. 2011. 3d is here: Point cloud library (pcl). In *2011 IEEE international conference on robotics and automation*. IEEE.

Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2018. SCALE-Sim: Systolic CNN Accelerator Simulator. arXiv:1811.02883

Joshua San Miguel, Jorge Albericio, Natalie Enright Jerger, and Aamer Jaleel. 2016. The bunker cache for spatio-value approximation. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load value approximation. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 127–139.

Satyabrata Sarangi and Bevan Baas. 2021. DeepScaleTool: A tool for the accurate estimation of technology scaling in the deep-submicron era. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–5.

D. Scharstein, R. Szeliski, and R. Zabih. 2001. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. In *Proceedings of 1st IEEE Workshop on Stereo and Multi-Baseline Vision*.

David Schor. 2018. NVIDIA's Xavier System-on-Chip, HotChips 30. https://www.hotchips.org/hc30/1conf/1.12_Nvidia_XavierHotchips2018Final_814.pdf

Min-Woong Seo, Myunglae Chu, Hyun-Yong Jung, Suksan Kim, Jiyoun Song, Junan Lee, Sung-Yong Kim, Jongyeon Lee, Sung-Jae Byun, Daehee Bae, Minkyung Kim, Gwi-Deok Lee, Heesung Shim, Changyong Um, Changhwa Kim, In-Gyu Baek, Doowon Kwon, Hongki Kim, Hyuksoon Choi, Jonghyun Go, JungChak Ahn, Jaekyu Lee, Changrok Moon, Kyupil Lee, and Hyoung-Sub Kim. 2021. A 2.6 e-rms Low-Random-Noise, 116.2 mW Low-Power 2-Mp Global Shutter CMOS Image Sensor with Pixel-Level ADC and In-Pixel Memory. In *2021 Symposium on VLSI Circuits*. 1–2.

Yakun Sophia Shao and David Brooks. 2013. Energy characterization and instruction-level energy model of Intel's Xeon Phi processor. In *International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 389–394.

Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 97–108.

Martin Simonovsky and Nikos Komodakis. 2017. Dynamic edge-conditioned filters in

convolutional neural networks on graphs. In *Proceedings of the 30th IEEE conference on computer vision and pattern recognition*.

J. Smith, G. Petrova, and S. Schaefer. 2012. Progressive encoding and compression of surfaces generated from point cloud data. *Computers & Graphics* (2012).

Nikolai Smolyanskiy, Alexey Kamenev, and Stanley T. Birchfield. 2018. On the Importance of Stereo for Accurate Depth Estimation: An Efficient Semi-Supervised Deep Neural Network Approach. In *Proceedings of the 31th Conference on Computer Vision and Pattern Recognition Workshops*.

M. Song, J. Zhang, H. Chen, and T. Li. 2018. Towards Efficient Microarchitectural Design for Accelerating Unsupervised GAN-Based Deep Learning. In *Proceedings of the 24th IEEE International Symposium on High Performance Computer Architecture*.

Sony. 2020. Sony to Release World's First Intelligent Vision Sensors with AI Processing Functionality. https://www.sony.com/en/SonyInfo/News/Press/202005/20-037E/.

Jonathan Dyssel Stets, Yongbin Sun, Wiley Corning, and Scott W Greenwald. 2017. Visualization and labeling of point clouds in virtual reality. In *SIGGRAPH Asia 2017 Posters*. ACM, 31.

Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration* 58 (2017), 74–81.

Xuebin Sun, Han Ma, Yuxiang Sun, and Ming Liu. 2019. A Novel Point Cloud Compression Algorithm Based On Clustering. *IEEE Robotics and Automation Letters* (2019).

Christer Svensson and J Jacob Wikner. 2010. Power consumption of analog circuits: a tutorial. *Analog Integrated Circuits and Signal Processing* 65, 2 (2010), 171–184.

Lech Świrski, Andreas Bulling, and Neil Dodgson. 2012. Robust real-time pupil tracking in highly off-axis images. In *Proceedings of the symposium on eye tracking research and applications*. 173–176.

Lech Świrski and Neil A. Dodgson. 2013. A fully-automatic, temporal approach to single camera, glint-free 3D eye model fitting [Abstract]. In *Proceedings of ECEM 2013*. http://www.cl.cam.ac.uk/research/rainbow/projects/eyemodelfit/

Richard Szeliski. 2010. *Computer vision: algorithms and applications*. Springer Science & Business Media.

Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, et al. 2021. Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 654–667.

Xiyuan Tang, Jiaxin Liu, Yi Shen, Shaolan Li, Linxiao Shen, Arindam Sanyal, Kareem Ragab, and Nan Sun. 2022. Low-power SAR ADC design: Overview and survey of state-of-the-art techniques. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2022).

D. Thanou, P. A. Chou, and P. Frossard. 2016. Graph-Based Compression of Dynamic 3D Point Cloud Sequences. *IEEE Transactions on Image Processing* (2016).

Federico Tombari, Samuele Salti, and Luigi Di Stefano. 2010. Unique signatures of histograms for local surface description. In *European conference on computer vision*. Springer.

H Tsugawa, H Takahashi, R Nakamura, T Umebayashi, T Ogita, H Okano, K Iwase, H Kawashima, T Yamasaki, D Yoneyama, et al. 2017. Pixel/DRAM/logic 3-layer

stacked CMOS image sensor technology. In *2017 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 3–2.

Christos Ttofis and Theocharis Theocharides. 2014. High-quality real-time hardware stereo matching based on guided image filtering. In *Proceedings of the 6th the Conference on Design, Automation & Test in Europe*.

Chenxi Tu, Eijiro Takeuchi, Alexander Carballo, and Kazuya Takeda. 2019. Point Cloud Compression for 3D LiDAR Sensor using Recurrent Neural Network with Residual Blocks.

Chenxi Tu, Eijiro Takeuchi, Chiyomi Miyajima, and Kazuya Takeda. 2016. Compressing continuous point cloud data using image compression methods.

Velodyne. 2017. HDL-64E: High Definition Real-Time 3D Lidar. https://velodynelidar.com/products/hdl-64e/

Velodyne. 2020. Velodyne Lidar Introduces Velabit. https://velodynelidar.com/press-release/velodyne-lidar-introduces-velabit/

Jianfeng Wang, Cha Zhang, Wenwu Zhu, Zhengyou Zhang, Zixiang Xiong, and Philip A Chou. 2012. 3D scene reconstruction by multiple structured-light based commodity depth cameras. In *Proceedings of the 37th IEEE International Conference on Acoustics, Speech and Signal Processing*.

Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. 2019b. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909.01315* (2019).

Wenqiang Wang, Jing Yan, Ningyi Xu, Yu Wang, and Feng-Hsiung Hsu. 2015. Real-time high-quality stereo vision system in FPGA. *IEEE Transactions on Circuits and Systems for Video Technology*.

Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E Sarma, Michael M Bronstein, and Justin M Solomon. 2019a. Dynamic graph cnn for learning on point clouds. *ACM Transactions on Graphics (TOG)* 38, 5 (2019), 1–12.

Zhimin Wang, Yuxin Zhao, Yunfei Liu, and Feng Lu. 2021. Edge-Guided Near-Eye Image Analysis for Head Mounted Displays. In *2021 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*. IEEE, 11–20.

David Weikersdorfer, Raoul Hoffmann, and Jörg Conradt. 2013. Simultaneous localization and mapping for event-based vision systems. In *International Conference on Computer Vision Systems*. Springer, 133–142.

Claus Weitkamp. 2006. *Lidar: range-resolved optical remote sensing of the atmosphere*. Springer Science & Business.

Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning Structured Sparsity in Deep Neural Networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*.

Neil HE Weste and David Harris. 2015. *CMOS VLSI design: a circuits and systems perspective*. Pearson Education India.

P. N. Whatmough, S. K. Lee, D. Brooks, and G. Wei. 2018. DNN Engine: A 28-nm Timing-Error Tolerant Sparse Deep Neural Network Processor for IoT Applications. *IEEE Journal of Solid-State Circuits*.

P. N. Whatmough, S. K. Lee, M. Donato, H. Hsueh, S. L. Xi, U. Gupta, L. Pentecost, G. G. Ko, D. Brooks, and G. Wei. 2019. A 16nm 25mm2 SoC with a 54.5x Flexibility-Efficiency Range from Dual-Core Arm Cortex-A53 to eFPGA and Cache-Coherent Accelerators. In *Proceedings of the 7th Symposium on VLSI Circuits*.

Paul N. Whatmough, Chuteng Zhou, Patrick Hansen, Shreyas K. Venkataramanaiah, Jae-sun Seo, and Matthew Mattina. 2019a. FixyNN: Efficient Hardware for Mobile

Computer Vision via Transfer Learning. In *Proceedings of the 2nd Conference on Systems and Machine Learning*.

Paul N Whatmough, Chuteng Zhou, Patrick Hansen, Shreyas Kolala Venkataramanaiah, Jae-sun Seo, and Matthew Mattina. 2019b. FixyNN: Efficient Hardware for Mobile Computer Vision via Transfer Learning. *arXiv preprint arXiv:1902.11128* (2019).

Mark Whitty, Stephen Cossell, Kim Son Dang, Jose Guivant, and Jayantha Katupitiya. 2010. Autonomous navigation using a real-time 3d point cloud. In *2010 Australasian Conference on Robotics and Automation*.

T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. 2003. Overview of the H.264/AVC Video Coding Standard. *IEEE Trans. Cir. and Sys. for Video Technol.* (2003).

Wikipedia. 2016. Pascal (microarchitecture). https://en.wikipedia.org/wiki/Pascal_(microarchitecture)

Felix Winterstein, Samuel Bayliss, and George A. Constantinides. 2013. FPGA-based K-means clustering using tree-based data structures. In *International Conference on Field Programmable Logic & Applications*.

Daniel Wong, Nam Sung Kim, and Murali Annavaram. 2016. Approximating warps with intra-warp operand value similarity. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 176–187.

Bichen Wu, Alvin Wan, Xiangyu Yue, Peter Jin, Sicheng Zhao, Noah Golmant, Amir Gholaminejad, Joseph Gonzalez, and Kurt Keutzer. 2018b. Shift: A Zero FLOP, Zero Parameter Alternative to Spatial Convolutions. In *Proceedings of the 31st IEEE Conference on Computer Vision and Pattern Recognition*.

Bichen Wu, Alvin Wan, Xiangyu Yue, and Kurt Keutzer. 2018a. Squeezeseg: Convolutional neural nets with recurrent crf for real-time road-object segmentation from 3d lidar point cloud. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*.

Wenxuan Wu, Zhongang Qi, and Li Fuxin. 2019. Pointconv: Deep convolutional networks on 3d point clouds. In *Proceedings of the 32nd IEEE Conference on Computer Vision and Pattern Recognition*.

Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 2015. 3d shapenets: A deep representation for volumetric shapes. In *Proceedings of the 28th IEEE conference on computer vision and pattern recognition*.

Yuan Xie and Jishen Zhao. 2015. Die-stacking architecture. *Synthesis Lectures on Computer Architecture* 10, 2 (2015), 1–127.

Han Xu, Ningchao Lin, Li Luo, Qi Wei, Runsheng Wang, Cheng Zhuo, Xunzhao Yin, Fei Qiao, and Huazhong Yang. 2021. Senputing: An ultra-low-power always-on vision perception chip featuring the deep fusion of sensing and computing. *IEEE Transactions on Circuits and Systems I: Regular Papers* 69, 1 (2021), 232–243.

Tiancheng Xu, Boyuan Tian, and Yuhao Zhu. 2019. Tigris: Architecture and Algorithms for 3D Perception in Point Clouds. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 629–642.

Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. HyGCN: A GCN Accelerator with Hybrid Architecture. In *Proceedings of the 26th International Symposium on High Performance Computer Architecture*.

Minhao Yang, Shih-Chii Liu, and Tobi Delbruck. 2015. A Dynamic Vision Sensor With 1% Temporal Contrast Sensitivity and In-Pixel Asynchronous Delta Modulator for Event Encoding. *IEEE Journal of Solid-State Circuits* 50, 9 (2015), 2149–2160.

Qingxiong Yang. 2014. Hardware-efficient bilateral filtering for stereo matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Xuan Yang, Mingyu Gao, Jing Pu, Ankita Nayak, Qiaoyi Liu, Steven Emberton Bell, Jeff Ou Setter, Kaidi Cao, Heonjae Ha, Christos Kozyrakis, and Mark Horowitz. 2018. DNN Dataflow Choice Is Overrated. arXiv:1809.04070

Xuan Yang, Jing Pu, Blaine Burton Rister, Nikhil Bhagdikar, Stephen Richardson, Shahar Kvatinsky, Jonathan Ragan-Kelley, Ardavan Pedram, and Mark Horowitz. 2016. A Systematic Approach to Blocking Convolutional Neural Networks. arXiv:1606.04209

Amir Yazdanbakhsh, Kambiz Samadi, Nam Sung Kim, and Hadi Esmaeilzadeh. 2018. GANAX: A Unified MIMD-SIMD Acceleration for Generative Adversarial Networks. In *Proceedings of the 45th ACM/IEEE Annual International Symposium on Computer Architecture*.

Praveen Yedlapalli, Nachiappan Chidambaram Nachiappan, Niranjan Soundararajan, Anand Sivasubramaniam, Mahmut T Kandemir, and Chita R Das. 2014. Short-circuiting memory traffic in handheld platforms. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 166–177.

Wang Yifan, Felice Serena, Shihao Wu, Cengiz Öztireli, and Olga Sorkine-Hornung. 2019. Differentiable surface splatting for point-based geometry processing. *ACM Transactions on Graphics (TOG)* 38, 6 (2019), 1–14.

Yuk-Hoi Yiu, Moustafa Aboulatta, Theresa Raiser, Leoni Ophey, Virginia L Flanagin, Peter Zu Eulenburg, and Seyed-Ahmad Ahmadi. 2019. DeepVOG: Open-source

pupil segmentation and gaze estimation in neuroscience using deep learning. *Journal of neuroscience methods* 324 (2019), 108307.

Christopher Young, Alex Omid-Zohoor, Pedram Lajevardi, and Boris Murmann. 2019. A data-compressive 1.5/2.75-bit log-gradient QVGA image sensor with multi-scale readout for always-on object detection. *IEEE Journal of Solid-State Circuits* 54, 11 (2019), 2932–2946.

Donghee Yu, Choong jae Lee, Myounkyu Park, Junghwan Park, Seungju Hwang, Joonhyung Lee, Sunghun Yu, Hyunjung Shin, ByoungHo Kim, Jong-Won Choi, et al. 2019. 14nm FinFET process technology platform for over 100M pixel density and ultra low power 3D Stack CMOS Image Sensor. In *2019 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 8–1.

Haibo Zhang, Prasanna Venkatesh Rengasamy, Shulin Zhao, Nachiappan Chidambaram Nachiappan, Anand Sivasubramaniam, Mahmut T Kandemir, Ravi Iyer, and Chita R Das. 2017. Race-to-sleep+ content caching+ display caching: a recipe for energy-efficient video streaming on handhelds. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*.

Haibo Zhang, Shulin Zhao, Ashutosh Pattnaik, Mahmut T Kandemir, Anand Sivasubramaniam, and Chita R Das. 2019b. Distilling the Essence of Raw Video to Reduce Memory Usage and Energy at Edge Devices. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*.

Jie-Fang Zhang and Zhengya Zhang. 2021. Point-X: A spatial-locality-aware architecture for energy-efficient graph-based point-cloud deep learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1078–1090.

Kuangen Zhang, Ming Hao, Jing Wang, Clarence W de Silva, and Chenglong Fu. 2019a. Linked dynamic graph CNN: Learning on point cloud via linking hierarchical features. *arXiv preprint arXiv:1904.10014* (2019).

S Zhao, Haibo Zhang, S Bhuyan, C Mishra, Z Ying, Mahmut T Kandemir, Anand Sivasubramaniam, and Chita R Das. 2020. Déjà-View: Spatio-Temporal Compute Reuse for Energy-Efficient 3600 VR Video Streaming. In *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture*.

Yin Zhou and Oncel Tuzel. 2018. Voxelnet: End-to-end learning for point cloud based 3d object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

Yangjie Zhou, Mengtian Yang, Cong Guo, Jingwen Leng, Yun Liang, Quan Chen, Minyi Guo, and Yuhao Zhu. 2021. Characterizing and Demystifying the Implicit Convolution Algorithm on Commercial Matrix-Multiplication Accelerators. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 214–225.

Yuhao Zhu, Matthew Mattina, and Paul N. Whatmough. 2018a. Mobile Machine Learning Hardware at ARM: A Systems-on-Chip (SoC) Perspective. In *Proceedings of the 1st Conference on Systems and Machine Learning*.

Yuhao Zhu, Anand Samajdar, Matthew Mattina, and Paul Whatmough. 2018b. Euphrates: Algorithm-SoC Co-Design for Low-Power Mobile Continuous Vision. In *Proceedings of the 45th ACM/IEEE Annual International Symposium on Computer Architecture*.